Starting Systems 3: Execution Counts: Software Performance & Optimization

A software article by Efron Amber Licht

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

Intro and the Performance Hierarchy of Needs

> *“There is no magic. There is only knowledge, more or less hidden.”*
>
> Gene Wolfe,
> **Book of the New Sun**.

Computers are physical machines that consume resources to create and operate. Using as few resources to create outputs from inputs as possible is what separates a ~~real~~ systems programmer from the dilettante. This is my third article on the fundamentals of systems programming and the first about the subject nearest and dearest to my heart: **software performance**. It can be read on it’s own and I’ll try not to make too many references to previous material.

Software is a tool to produce output from input. More efficient, “high-performance” software produces that output **cheaper, faster, or more reliably, with the minimum input possible**. Doing so is easier said than done, but we can split the study of software performance into three parts: knowing your performance and finding bottlenecks (measurement), designing a system without any critical performance mistakes (architecture), and fixing the problems you can (optimization). We’ll cover all three, roughly in that order.

Note to the reader

This article is very long, nearly 20k words at my last count. If you're just interested in the practical bits, check out the guide to pprof and the practical optimization and benchmarks in the back half: I won't judge too much. Most headings are self-links and you can click on the line numbers on any section of code to get a sharable link, so it should be easy to save your place and come back later, and I've provided a table of contents below.

Recap/Overview

- In Part 1: Programmers Write Programs, we built a series of software tools in order to investigate the layout and behavior of programs.
- In Part 2: Your Program and the Outside World, we investigated the fundament,*als of the UNIX programming environment - command-line arguments, environment variables, syscalls, files, and more, leading up to a basic command-line shell.
- Here in Part 3: Execution Counts, we'll cover software performance. This article will try to cover the whole gamut, from basic concepts, through measurement, metrics, and profiling, to architecture, and finally hitting real optimization and benchmarks.

Measurement Basics

A systems programmer **measures** and does not guess. The core of software performance is **measurement** and an understanding of the system. Measurement comes in many kinds, but they come in a specific order: first **tests**, then **metrics**, **profiles**, and **benchmarks**, *in that order*.

- **Tests** tell you your programs works. Fast and wrong is still wrong. Testing is outside the scope of this article, but it's always worth emphasis. (It is extremely easy to have the fastest benchmark by solving the wrong problem.)
- **Metrics** tell you what resources your program is actually consuming and which ones are limiting. Don't spend time fixing problems that aren't problems. If you're

only using 2% of your CPU and 90GiB, worry about the RAM, not the CPU. We'll cover metrics only a little in this article - I hope to have one out soon about that specifically.
- **Profiles** tell you where in your program those resources are being used, localizing the problem. The patch on a hose should go where it's leaking. While it's fun to optimize every corner of your program, sadly, the systems programmer has limited time and must focus on the priorities, and human beings are very bad at looking at a million-line codebase and figuring out where the performance problems are.
- **Benchmarks** confirm your **new** solutions are actually better than the old one.

Optimization Basics & the Kitchen Counter

Once you've found a problem, how do you fix it? There are only a handful of basic techniques to improve the performance of any task, and they don't only apply to computers. In fact, we do them all the time without realizing they are 'optimizations': human being naturally try to avoid wasted effort in the physical world. For example, if I want to eat popcorn, I don't get up from the couch, get in my car, drive to the store, buy a single popcorn kernel, go home, pop it over the stove, take it to the couch, sit down, eat the popped kernel, then repeat. I buy many bags of microwave popcorn at the store at the same time (batching), at the same time as I purchase other items (optimizing access patterns), store them in my cabinet (caching), pop them in the microwave while I do something else (scheduling/pipeline optimization), then take the whole bag to the couch (delicious). Sadly, in software, this kind of horrifically inefficient design is the rule rather than the exception.

> We'll call this kind of bad design "pessimized", to steal a term from Casey Muratori. Because performance is limited by bottlenecks, it is generally more important to avoid foolish pessimization than to chase optimization.

Table: Basic Optimization Techniques

The following table summarizes the basic techniques of optimization using both software and kitchen examples.

casual description	technical description	software example	kitchen example
do less work	algorithmic optimization	binary versus linear search	dicing an onion 'properly' - N+M cuts for NxM diced bits.
do the same work faster	micro-optimization	loading from cache vs loading from disk; just buying a faster computer;	using food processor for small cuts & cleaver for large cuts
do tasks physically close	optimizing access patterns	struct of arrays vs array of structs	chopping all onions, then all peppers, etc;

casual description	technical description	software example	kitchen example
together at the same time			
do it using pre-prepared materials	preprocessing	serving prebuilt static webpages pages vs forming dynamic ones; caching	‘mise en place’; buy pre-diced garlic;
do work at the same time	parallelization	web server serving individual clients on separate threads	running multiple burners
do it in an order that involves less waiting	scheduling/ pipeline optimization	start dependencies in a background thread; nonblocking IO	pre-heating the oven - putting the turkey in the oven first b/c it takes the longest; having guests serve themselves; bring dishes as they are ready
do it without waste	efficiency	taking advantage of side effects; eliminating unnecessary code	using scraps, organs, and marrow for soup & fats for schmaltz
synchronize less	pipeline optimization	reduce number of services; reduce number of network calls; reduce mutex contention	remove an ingredient; prepare ingredients;separately
do nothing	feature rejection	CLOSED: WON’T DO	skip lunch

> Each of these could be the subject of an article or a book on their own and are worth your time. First and most importantly, a proper understanding of algorithms and data structures is essential for performant programming. The combination of the hashmap and resizable array can take you a long way, but for some problems there’s simply no substitute for the right data structure. (Happily, most of them can be built out of combinations of hashmaps and resizable arrays). I learned off CLRS, which is a good reference for those with a formal mathematics occasion, but pretty tough for a first approach. If you don’t have a formal CS background, try a couple of different books and see which ones gel with you.

Why Performance?

> *A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.*
>
> - Antoine de Saint-Exupéry
>
> Yes, I quote this in every article.

I hear all the time that performance doesn't "really" matter. I strongly disagree. Here are *some* motivations for software performance.

- **Aesthetic**: Beauty & elegance comes from doing more with less.
- **Corporate**: Software that's faster, cheaper, and more responsive is more fun to use and will perform better in the marketplace.
- **Ethical**: A professional *always* does a good job: that's what makes them a professional.
- **Financial**: `profit = revenue - cost`. Reduce cost, increase profits.
- **Hedonic**: Optimizing is *really fun*. Number go up! (Or even better, down. Play Factorio or SHENZEN I/O if you don't believe me).
- **Moral**: Needless waste is a kind of theft from others and from the future.
- **Operational**: Some operations are simply *not possible* with poor software. A game that cannot render frames fast enough simply does not work.
- **Practical/Professional**: Practice makes perfect. If you don't practice making *ordinary* software efficient, you won't have the skills when it really matters.
- **Professional Honor**: Bad software shames all programmers. We should feel ashamed of the dismal quality of most software and how much of our vast computing resources are wasted.
- **Strategic**: The world (and the subset of resources in it available to you) finite. Using resources now means they won't be available for more important things later.

The Lay of the Land: Concepts, Resources, Tradeoffs

Let's back all the way up. What do we mean when we talk about a 'high-performance' machine? What specifically did we mean by "cheaper" and "faster"?

> Using 'machine' in the abstract sense. Usually this is a software program, but it could be a physical machine, an algorithm for sweeping the floor, whatever. It's a Thing That Solves A Problem.

When we talk about the performance of a machine, we mean it's ability to take inputs and convert them into outputs. A 'high-performance' machine is one that excels in one or more of the following categories:

- can handle all incoming traffic (high **bandwidth**)

- responds quickly to new input (low **latency**)
- uses few inputs to produce outputs (high **efficiency** / low waste)
- continues to function over time in a predictable way, even under unpredictable load (high **reliability**)
- and without nasty side effects (**purity**)
- as simply as possible (**simplicity** / elegance)
- without doing anything truly barbarous (**integrity**)

These concepts don't just apply to computers: *all* of our technology can and should be judged on this criteria. As such, for each I'll try and have a couple of examples in the tradition of my grandfather

Latency: How fast?

The time from input to output for an *individual* input.

Human beings are extremely latency-sensitive machines. Lower latency should always be a priority, *especially for interactive software*. Nothing kills fluid interaction with computer like input latency. If you don't believe me, try switching your monitor to 15fps and sending me an email without a typo.

> From Dan Luu: "Input Lag"
>
> Latency matters! For very simple tasks, people can perceive latencies down to `2 ms` or less. Moreover, increasing latency is not only noticeable to users, it causes users to execute simple tasks less accurately. If you want a visual demonstration of what latency looks like and you don't have a super-fast old computer lying around, check out this MSR demo on touchscreen latency.
>
> For my dishwasher, my *minimum* latency is the cycle time - two hours. Most dishes take much longer than that to clean, since they're waiting in the racks for longer.
> There's also a minimum latency to load or store any item from the dishwasher, since you have to open up the dishwasher and reposition the racks. You can somewhat cut this latency by loading or retrieving multiple elements at a time.

- **p50 latency** - median time to process an input
- **p99, p99.9, p99.99 ('tail') latency** - how long the most unlucky 1%, 0.1%, etc of inputs take to process.

Latency is usually the least-negotiable performance characteristic. No amount of arguing or clever engineering can make a packet go faster than light, so don't send a packet on a trip without a purpose.
Since you can't fix latency later, it merits the most attention during the design phase of any performance target.

Improving Latency in Software

Which computer operations have low latency?

The following numbers - taken from the famous "latency numbers every programmer should know" and lightly edited - give an idea of these relative latencies. While hardware changes, the *relative* speeds are still a good guide.

Latency Table (circa 2012)

what	time	commentary	—	
mathematical instruction	0.1 ns			
jump (predicted, within cache)	1 ns			
L1 cache reference	0.5 ns			
Branch mispredict	5 ns			
L2 cache reference	7 ns	14x L1 cache		
Mutex lock/unlock	25 ns			
Main memory reference	100 ns	20x L2 cache, 200x L1 cache		
Context Switch or System Call	1000 ns			
Compress 1K bytes with Zippy	3,000 ns	3 µs		
Send 1K bytes over 1 Gbps network	10,000 ns	10 µs		
Read 4K randomly from SSD*	150,000 ns	150 µs	~1GB/sec SSD	
Read 1 MB sequentially from memory	250,000 ns	250 µs		
Round trip within same datacenter	500,000 ns	500 µs		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 µs	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 µs	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 µs	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000 ns	150,000 µs	150 ms	

TLDR: **Do things in-memory and in-process where possible. Program and computer boundaries are extraordinarily expensive**.

Purity: Few if no side effects.

A "pure" machine or program converts it's input into outputs without having a negative impact on the world around it. "Side effects" are other ways of thinking about this. Impurity is something that you should only tolerate if you're getting something valuable from it. Caches are the canonical example of sacrificing purity for latency and bandwidth.

> My dishwasher has a number of side effects, including unwanted noise, venting hot water vapor into my kitchen, and lowering the pressure of hot water available elsewhere in the house (more about that one in a second).

Efficiency: How Cheap?

An efficient solution consumes fewer inputs to produce the desired outputs. Inputs can either be totally consumed by the process (like turning wood into paper) or temporarily occupied (like the physical space on the conveyor belt in a factory).

> Using less flour to make bread is one kind of efficiency, as is using less fuel, as is occupying a smaller oven. For a computing task, a good analogy might be clock time, electricity, and RAM.
>
> More efficient dishwashers use less water and power. A very large kitchen might be best served by having multiple dishwashers of different sizes and shapes for different kinds of loads.

The best kind of efficiency comes from **less**: less code running on fewer machines in less time costing less power. Sometimes you can gain efficiency by a kind of asymmetrical tradeoff, where, for example, a little bit of RAM can you a whole lot of CPU time. More on that later.

Bandwidth: How much?

The numbers of inputs and outputs that can be processed over a time interval under the given conditions, usually assuming optimal packing of inputs. How exactly you measure bandwidth can get real complicated real fast.Usually, there are multiple kinds of bandwidth - /machine/minute, for example - and you need to figure out which one is the limiting factor, the bottleneck.

> The dishwasher has 2 racks, each of which are subdivided into a 12x6 grid, about big enough for a water glass each. `12*6*2` = `12*12` = 144 "glases per

cycle". The cycle conveniently takes about 140 minutes, so we'll say it's a bandwidth of 1 GPM 'glass per minute'.

A more through approach would involves measuring the insides of the racks to figure out the actual surface area of the racks.

Of course, in practice, we don't pack dishwashers with uniform and identical dishes, and if you pack a dishwasher too tightly it doesn't work.

Simplicity: Do I understand it?

Simpler solutions are usually faster and more efficient, and they're easier to change later if your requirements do. Be wary of 'free' performance you don't fully understand: i.e, other people's libraries that look good on benchmarks. Only take on additional complexity **if it does something for you.** Only take on additional complexity you don't understand if you have no other choice.

I chose a dishwaster with a **garbage disposal** because it means I don't have to clean a trap. I did *not* choose one with an "AI wash" because that sounds like nonsense and is additional complexity *without* a known return.

"The Lay of the Land": Problem: "Optimizing your Daily Life"

Write down the ordinary chores that you do day to day (cooking, sweeping, washing dishes, folding laundry, etc). What kinds of tradeoffs do you make in these day to day? Can you think of any obvious improvements to the algorithms you use here? Hint: try taping yourself actually doing one of these tasks, or watching a friend do them. Most people are pretty inefficient. Try your new approach and compare.

Reliability: Will it keep working?

The bandwidth and efficiency of a broken machine is zero and it's latency is infinite. A solution that only works on Tuesdays before noon is no solution at all.

My dishwasher is not perfectly reliable. For one thing, it's on the same pipes as the rest of my hot water, a relatively old system. The system can handle two hot showers OR a dishwasher, but not both: that kind of load will hurt all three systems. In other words, the impurity of one system has knock-on effects on the reliability of another.

The **garbage disposal** of the dishwasher is an additional point of failure, but the dishwasher contains (smaller) trap as a failsafe. Worst comes to worst, I can wash by hand.

One kind of reliability that's not intuitive is "graceful degradation under load".

> Compare the behavior of cars and trains during rush hour or other extreme loads. A drain may handle a large amount of rain OK but back up during a storm.

Integrity: But why did you do it?

Our goal is to produce software that **has** high-performance. Unfortunately, the world is filled with plenty of people who will settle for software that **appears high-performance**.

It is incredibly easy to cook benchmarks to favor your software. Once any measurement becomes a target - that is, there is some kind of reward for being "the best" (usually money or promotion, but sometimes just attention) - there will be a temptation to cook the books to achieve that reward without earning it. Even in situations where there's no plausible reward, some people just cheat on benchmarks for the hell of it.

At an even more basic level: **software should change to become better rather than worse.** It's easy to add pointless layers to avoid a political fight at work, but it's wrong, and it will lead to bad software. A real programmer is loyal to the truth.

Performance Concepts: Tradeoffs

While we would like to have all of the above, usually you have to choose some balance or proportion between them.

Latency, **Bandwidth**, and **Efficiency** are always in tension. Consider scheduling for commercial air flights. It's **fastest** to have lots of small flights taking off all the time, but this would stress the airports' scheduling systems to their latest, so they prefer fewer, larger flights, which are cheaper for the airline to run and get more people around faster. Similarly, it's far more efficient to bundle together `INSERT` statements to a database, but waiting to collect multiple rows will hurt your turnaround time for the individual ones.

Efficiency and **Reliability** also often trade off. The most efficient machines have no waste, but they also have no headroom. To use commercial airline scheduling as an example, most flights have some cancellations, which is another way of saying they don't use all of their seats. An airline can get better *efficiency* by overcommiting the seats on the average flights, but when a particular flight happens to fill all it's seats, *someone* is going to miss their flight, which will kill latency and efficiency.

Bandwidth and **Complexity** often trade off. A clever programmer with good knowledge of SIMD can get incredible throughput at the expense of portability and confusing the heck out of whoever has to deal with their code later.

Complexity and **Reliability** trade off, too. If and when a complicated system breaks down, it can be almost impossible to get at and/or fix.

Integrity trades off with everything else. Once your integrity goes, everything else will follow.

> "The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair".
> **Douglas Adams**

Software Architecture

Software Architecture: Understand your problem

When we think about software performance, this usually seems like a very low-level endeavor, best suited for grizzled C++ warriors and the last remaining assembler wizards of the high hills. While this has some merit to it - you'll never make truly great software without a decent sense of how the data flows through your machine - it is secondary to the decision of what software to build to begin with.

After all, when you are searching for gold, good technique helps, but the first step is to **pick a site with gold in it**.

First understand your problem - that is, what your inputs are, how they are distributed, and the expected outputs and performance characteristics, *then* get to the implementation details.

> Of course, your classic computer science data structures and algorithms loom over this whole process and should always be in the back of your mind, since no amount of clever programming or sophisticated architecture will turn an `O(n^2)` into an `O(log(n))`. On the other hand, whether that's a problem depends on how big you expect `n` to be.

Think about who is using your program, what they want, where, and why. This will inform you of your priorities and cutoffs for latency, throughput, etc.

For an example, let's talk about how ordinary people in new york city transport themselves and other objects from Brooklyn to Manhattan.

- **Commuters themselves** are high volume, moderate latency, and low cost, and are best served by the subways or walking.
- A **grand piano** can't fit in the subways and is in no rush: the cheapest and best way to move it is by truck.
- A **heart transplant** may be best sent via helicopter: the enormous cost and miserable bandwidth are justified by the extremely low latency of literal flight.

Similarly, the correct program for one situation may be completely wrong for another even if they solve the same problem from one point of view.

Software Architecture is hard to teach from general principle, so I'll tell you a story taken from my real life experience. That will segue nicely into learning about profiling, benchmarking, and 'real' low-level optimization.

Welcome to 'Cable Time'.

Software Architecture: 'Cable Time': Intro

You're reading this on the internet right now, which means you have some kind of Internet Service Provider. If you're reading it on a computer in the United States, like I am writing it, that internet service is likely carried part of the way as a *encoded electrical signal* over a shielded cable ('coaxial cable').

That coaxial cable carries network traffic to a Cable Management Termination System (CMTS) somewhere near you, which then talks to a bigger router, which talks to the internet.

That is, the current network topology for a cable internet subscriber might look something like this:

diagram poor scaling

> The history of telecommunications is far too big a subject to get into here, but Jon Bois' Fool Time is a wonderful time and as good of a place to start as any. ('Cable Time' is no coincidence.)

That electrical signal has *noise* - that is, some of what you get on the receiving end is *not what you put in*. **Coaxial** internet cables do a remarkably good job of shielding that wire, but no electrical system is perfectly noise free. Happily, the noisy-channel coding theorem tells us that for any noisy channel, you can construct an encoding that will reliably communicate data.

> Famously in Claude Shannon & Warren Weaver's *The Mathematical Theory of Communication*, which may be the most important mathematical result in computing history. Great book and surprisingly readable.

You have to know how noisy it will be ahead of time. If it turns out to be quieter than you expected, no data is lost, but you'll have sent at less than your maximum throughput. If it turns out to be noisier than you expected, you'll lose some of the data, which is usually much worse than missing out on a little throughput.

As it turns out, this noise is not constant. The type and intensity of that noise depends on what's actually happening in real life in and around the cable. High demand in the local electrical grid can make the cables 'noisy', and so can thunderstorms.

> Wires with electricity running through them are tiny electromagnets, after all.

Physical wear on the shielding around the cables can fiddle with it, too, not to mention damaged shielding, local moisture, insect activity. There are seasonal and chronological patterns to this data. Less load on the system means less noise: as people go to bed, they turn their lights and computers off, electrical demand drops off, and the local cable internet grid literally quiets down.

The most efficient service will come from using a digital-to-analog encoding that is *just loss robust enough for the local conditions*. You can use a more efficient and less robust encoding in 'quiet' areas and fall back to the heavy-duty stuff so that your customers in the 'loud' areas still get reliable internet, even if it's slower. To properly choose those encodings, you need good data.

The cable modems - and the CMTS (**C**able **M**odem **Termination S**tation/**S**ystem) they connect to - the big router on the other side - has all sorts of internal statistics on packet latency, % drop, local electrical noise, etc.

As you collect more data, you might be able to make models that can detect (or even predict!) these kinds of problems, mitigating them before anyone calls you to complain.

An ISP I worked for used to have an API that let their services query the Cable Modems and CMTSes for this kind of local data. Each of their own teams could ask for the data they needed, when they needed it, directly from the modems.

Diagram: Original self-serving cable modem metadata

For an individual service, this was trivial to implement and to understand.

diagram: trivial self-serve metadata flow

> Note: from now on, I will just say 'modems' and mean both cable modems and CMTSes.

At first, this worked well: it was self-serve and straightforward. As more and more services started asking for more and more data, the pretty graph above started to look a little like this, which was not so good:

Diagram: Poor scaling as many services request the same data

diagram: poor scaling as many services request the same data

The ISP had *hundreds* of services and millions of modems, and many of these services were making dozens of simultaneous calls. **Soon the ISP found out that their performance was degrading because the metadata traffic - was starting to overwhelm the ordinary internet traffic!**

This well-intentioned performance 'solution' had become a problem. Well, at least four problems:

1. Too many services were asking the modems for data.
2. The modems were being polled too often.
3. Services were asking the modems to do too much work - that is, asking the modems to collect and send information that they didn't actually need.
4. These problems would cascade - as problems with the network started to pile up, the downstream services would ask for *even more metadata* to try and figure out what was going on, making the problem even worse.

'Cable Time': Evaluation

Let's take a step back and think about our evaluation criteria. What are the different groups here, and what do they want?a

- **cable modem customers** want
  - reliable service
  - that's as fast as possible
- **services** (consumers of the metadata) want their metadata
  - from a relatively simple API
  - as quickly as possible
  - at a rate that scales sublinearly with the number of modems they want to collect from
- the **boss** wants:
  - the above
  - as cheaply as possible
- t **engineers** want:
  - To make the simplest system they can that satisfies all of the above, since if it gets too complicated nothing will work at all.

Let's turn this into more concrete, measurable goals.

- We want to be able to provide 'fresh' metadata to as many services as ask for it. That is, if a service asks for metadata, that metadata should have been collected by the modem less than some time `MaxAge` ago (where we choose `MaxAge` - say, 30 seconds).
- We want to *minimize* the amount of work done by the modems. **This means collecting at most twice per** `MaxAge` **cycle.** (It can't be at most once, since any jitter at all would break that guarantee). Not twice per consumner per `MaxAge`, twice at all.
- In the case of failure, we want to *partially* degrade and we want that degradation to happen on the 'serving metadata' layer, not on the 'serving customers internet' layer.
- We want a degree of overhead with consumer hardware. Assuming we are successful and the parent ISP is successful, there may be more services and modems down the pipe.

All three of these indicate that *collecting* and *serving* the data should be separate services.

Without compromising on the above, we want to do this as cheaply as possible. "Cheap" means

- using fewer, weaker machines where possible
- storing as little as possible
- sending as little network traffic as possible
- keeping demand relatively steady where you can pay for it ahead of time and save money. 'bursts' of demand are dangerous if you're self-hosting and expensive on 'elastic cloud' providers like AWS, etc.

Pricing Heuristics

A good way to get a sense of these relative costs - in dollars - is to look at AWS's ec2 on-demand pricing page. Don't pay too much attention to the specific numbers, but hold one parameter still and go 'up' and 'down' in various capabilities to get a sense of the % increase in price for, say, 64GiB of RAM vs 128GiB.

'Cable Time': Solving Evaluation Criteria via Design

With these criteria in mind, *now* was the time to design a high-level solution to the problem. There are probably a variety of ways to solve this, but they settled on a paired set of services - `cablewatch` and `cableshow` - and a database to serve as a cache between them.

- `cablewatch` asks the modems for metadata and caches them in `cabledb`.
- `cableshow` reads the db and distributes the cached data to services that ask for it.

The following graph compares the two architectures:

Graph: `cablewatch & cableshow` vs direct self-serve

diagram comparison: old vs new flow

Let's talk about how this design attempts to solve the problems listed above.

1. **Too many services were asking the modems for data.**
    - Only `cablewatch` can collect the data, which is stores for later retrieval via a relational database.
2. **Modems polled too often.**
    - `cablewatch` collects *all* of the data at regular intervals and never does more, regardless of how insistent the downstream consumers are.

3. **Services were asking for too much data.**
    - Customers must ask `cablewatch` for *exactly* what they want (the implementation used `graphql`, but there's a variety of ways to do this)_.
    - `cablewatch` only retrieves *exactly* that data rather than filtering too much. Happily, relational databases are very good at this.
4. **Cascading Failures**
    - `cableshow` must be independent of `cablewatch`, so failures do not 'spread' to the modems themselves.
    - `cableshow` should be able to scale up independently as load increases.
    - `cableshow` should rate limit it's consumers to force them into good behavior: it's better to drop some requests and be able to serve others than be able to serve no one at all.

'Cable Time': Design decisions for the new system @ the 'Operational' level

We are now pretty confident that the solution above will solve our problem at the high level, but it's not as simple as spinning up two new programs and a database - for one thing, *what language should those programs be in*, and why is it Go? What database(s) should we use? Why? How and where should we deploy?

'Cable Time': Design Decisions: 'Problems':

- What languages and/or databases would *you* use for this problem? Why?
- Reflect on the design process for the last major program or library you wrote. Was the process correct? What did you do right? What did you do wrong?

Then there are more specialized questions having to do with this specific problem:

`cablewatch`:

Should this be one system or many? if 'many', how many, **how do we partition the load?** How do we survive `cablewatch` going down?

`cableshow`:

Assuming this is many instances, how do we balance traffic? how do we detect 'bad actors'? what's our rate limiting and mirroring strategy? how do we recover after outages?

`cabledb:`

This is a *ton* of data - many, many TiB, too big for individual hard drives or even servers. this means a distributed system of some kind. **How do we partition the load to be retrieved across the database?** How will this partitioning impact read times for consumers?

'Cable Time' - Partitioning

Part of the solution involved partitioning the dataset via the Object Identifier (OID). (More on these later, it'll be important - but not quite yet).

Each individual cable modem or CMTS has a unique OID, so at most one `cablewatch` instance and at most one `cabledb` shard will be responsible for any given modem or CMTS (assuming we don't over-replicate for reliability). This has the same amount of total *write* load on both the DBs and the cable modems/CMTSes as the original system, and should 'spread out' the read load. This protects the system somewhat, but increases the total amount of work for reads and may cause additional latency if what used to be a single read on the DB now must become multiple reads spread across different databases.

With partitioning in mind, the solution looks like this:
partitioned architecture: cablewatch and cableshow

> Proper partitioning can make a huge difference in read latency. Ideally, partitions should be arranged both so the usual read patterns match the distribution of the data - but this may cause spikes in load on particular nodes. If you're interested in questions like this, Martin Klepmann's Designing Data-Intensive Applications is the best book I know of. It just came out with a second edition which I have not yet read, but I have to imagine it's great.
>
> That said, partitioning is one of those things that only starts mattering at very large scales, and even then, most of the time you don't need it. **Don't add partitioning - or any kind of complex trick** to a system that doesn't need it

This was a solid design and it was time to move on to implementation.

'Cable Time' - Implementation & Real Operational Costs.

The ISP developed according to this plan, building a new team of engineers to run it, including a mid-level named 'Efron Licht'. (This is where I entered the story). The new architecture worked pretty well and things went more or less according to plan, always a pleasant surprise in software. However, `cableshow` was RAM-hungry, with steady-state RAM consumption of about 90GiB and peaks of up to 130GiB. Our cloud provider had no

150GiB option, jumping straight from `128GiB` to `256GiB` machines, which were extremely expensive at the time - expensive enough that if we could cut that ram consumption, it could save us hundreds of thousands a year in cloud costs.

> We were actually *under budget* in cloud costs compared to other departments. ISPs spend money like water.

I got obsessed with the problem. Was there some way to knock down memory usage without compromising on other factors? A **memory profile** of `cableshow` might show us the bottleneck that was costing us so much money. If you're using Go, the best way to do this is `pprof`.

profiling with `pprof`: a practical guide

what is pprof?

`pprof` is a profiler/analyzer originally implemented in C++. It does the *analysis* of profiles, not the collecting. That's the job of the go runtime (specifically, `runtime/pprof`), which collects metrics and can output them in `pprof`'s preferred format.

- C++ source: github.com/google/pprof
- tool official documentation
  You can use that tool to download and analyze those profiles via `go tool pprof`.

pprof outside go

pprof is most associated with Go and C++, but you can use it in other languages via libraries like `pypprof` (python) and `jpprof` (java). Give it a shot!

Overview of Go pprof libraries

`runtime/pprof`

This library contains the low-level tools to enable, disabled, and export performance profiles. Usually, you prefer:

`net/http/pprof`

This library automatically collects performance profiles and lets you run a HTTP server that both serves that data out for clients (like `pprof` itself) AND provides a nice

interactive webserver you can visit in your browser, usually at `localhost:6060/debug/pprof`.

go test: flags

The `go test` tool contains built in support for exporting `pprof` profiles. We'll do this later.

pprof: guides and resources

- julia evans: profiling go programs with pprof
- russ cox & shengou ma: go.dev/blog/pprof: profiling go programs
- `go tool pprof --help` is always a good place to look, but is extremely terse.

pprof: example: profiling gitlab.com/efronlicht/ewaste

We need something to profile. Since this is a tutorial, it would be nice if we had an example where we knew ahead of time where the performance hotspots would be. Happily, I already have the perfect program for this: `ewaste` - a program to **deliberately waste resources**. `ewaste` wastes CPU time by finding primes, memory by allocating and then writing (so as to force a page fault and 'really' allocate), and disk by opening, writing, and syncing files. I updated `ewaste` to run a pprof server at `localhost:6060` so we have a simplified environment to learn about `pprof`.

> This may sound crazy, but it's useful to try and reproduce errors in other programs. IIRC I built it to find a weird interaction between NVIDIA drivers, Docker, and RAM.

Let's see what happens when we fire up an `ewaste` interactive program.

sh 1: ewaste interactive session

```
1  #!/usr/bin/env bash
2  $ ewaste
3  serving pprof at http://localhost:6060/debug/pprof
4  CPU Profile:
5          go tool pprof http://localhost:6060/debug/pprof/
   profile
6  Heap Profile:
7          go tool pprof http://localhost:6060/debug/pprof/
   heap
8  entering interactive mode. type `help` for usage
   instructions.
9  (ewaste)> help
```

```
10  HELP: try cpu|mem|disk followed by a number and optional
    unit. For example: `cpu 4`, `mem 512 mib`, `disk 10 gib`
11  (ewaste)> cpu 3
12  cpu: 3 cores, mem: 0 allocs,   0 B, disk: 0 files,   0 B
13  (ewaste)> mem 2 gib
14  cpu: 3 cores, mem: 1 allocs,   2 GiB, disk: 0 files,   0 B
```

We should now have three cores busily finding primes and a big block of wasted memory. Let's go find them. First, we'll run a cpu profile, which collects statistical data on what instructions are executed over a period of time.

`pprof`: cPU profile

`pprof`: cpu profile: collecting the profile

`go tool pprof [-seconds n]` will automatically fetch the CPU profile and enter an interactive session.

```
 1  #!/usr/bin/env bash
 2  # profile for 5 seconds, then open an interactive pprof
    session.
 3  $ go tool pprof -seconds 5 http://localhost:6060
 4
 5  Fetching profile over HTTP from http://localhost:6060/
    debug/pprof/profile?seconds=5 # this is the actual HTTP
    request to download the profiling file.
 6  Please wait... (5s) # profiling happens here
 7  Saved profile in /home/efron/pprof/
    pprof.ewaste.samples.cpu.002.pb.gz
 8  File: ewaste
 9  Build ID: 7029da92c5ec77a7e6134f7ce2adb8feb4f8b467
10  Type: cpu
11  Time: 2026-03-03 16:17:36 PST
12  Duration: 5s, Total samples = 14.89s (297.53%) # note that
    we have more than 100%! 100% means 100% of ONE processor.
    modern computers have many processors.
13  Entering interactive mode (type "help" for commands, "o"
    for options)
```

`pprof`: cpu profile: finding expensive functions with `top`

We'll use the `top` subcommand to find the functions that are the biggest consumers of CPU cycles. In an interactive pprof session, `top [n]` will show us the `n` highest users of CPU time.

```
1  # still in interactive session...
2  (pprof) top 20
```

```
Showing nodes accounting for 14880ms, 99.93% of 14890ms
total
Dropped 1 node (cum <= 74.45ms)
      flat  flat%   sum%        cum   cum%
    8320ms 55.88% 55.88%    14890ms   100%  main.wasteCPU
    2660ms 17.86% 73.74%     4050ms 27.20%
runtime.chanrecv
    1620ms 10.88% 84.62%     2080ms 13.97%  context.
(*cancelCtx).Done
    1390ms  9.34% 93.96%     1390ms  9.34%  runtime.empty
     460ms  3.09% 97.04%      460ms  3.09%  sync/atomic.
(*Value).Load (inline)
     430ms  2.89% 99.93%     4480ms 30.09%
runtime.selectnbrecv
```

pprof: understanding top

- `cumulative (%)` - total percentage of time spent in that function, rounding to the nearest 0.1%.
- `cum (ms)` - sum of total time spent in that function *for all CPUs*. this is why a 5s sample can have a function which takes nearly 15s - of cumulative time.
- `flat (%)` - percentage of time spent in that function, but not it's subfunctions. that is, the percentage of CPU time spent with the program counter literally executing a statement inside that function.
- `sum%` - `flat %` + (`prev sum%` or 0)

Looks like `wasteCPU` and it's subfunctions take up all the time, as we'd expect. However, `wasteCPU` itself is only taking slightly more than half. Looks like the progran's taking a good amount of time calling `context.Done()` and receiving from a channel.

Knowing which function calls are taking the time is handy, but it would be better if we could localize further - say, find the exact lines where the processor is spending it's time. Happily, we can. `list <pattern>` in an interactive `pprof` session will show us where time is being spent in functions that match the pattern.

pprof: breaking down function costs with `list`

The `list` subcommand outputs annotated source code. Let's try it out:

```
// still in previous interactive session
(pprof) list wasteCPU
Total: 14.96s
ROUTINE ======================== main.wasteCPU in /home/
efron/go/src/gitlab.com/efronlicht/ewaste/main.go
     7.98s     14.94s (flat, cum) 99.87% of Total
         .          .    188:func wasteCPU(ctx
context.Context) []int {
```

```
 7         140ms      140ms     189:   primes :=
    append(make([]int, 0, 4096), 2, 3, 5, 7, 11, 13, 17, 19,
    23)
 8              .          .     190:SPIN:
 9              .          .     191:   for {
10              .          .     192:           select {
11         2.17s      9.13s     193:           case
    <-ctx.Done(): //   runtime.chanrecv(context.
    (*cancelCtx).Done)
12          10ms       10ms     194:                   return
    primes
13              .          .     195:           default:
14         290ms      290ms     196:                   for i :=
    primes[len(primes)-1]; i < math.MaxInt32; i += 2 {
15         1.10s      1.10s     197:
    for _, p := range primes[1:] {
16         4.25s      4.25s
    198:                                   if i%p == 0
    { // not prime
17          20ms       20ms
    199:                                       continue
    SPIN
18              .          .
    200:                                   }
19              .          .     201:                       }
20              .          .     202:
    primes = append(primes, i)
21              .          .     203:                   }
22              .          .     204:
```

The cost breakdown looks something like this:

- 61% time spent checking for cancellation (line 193)
- 38% time spent actually looking for primes (lines 196..=199)

As it turns out, `wasteCPU` is a prime-finding routine that spends less than half of it's execution time actually looking for primes!

> go trivia: the built-ins `append` and `make` look like functions, but they're more like special macros, so their flat and cumulative time are identical. On the other hand, `case <- EXPR` looks like a control flow expression but really invokes the `runtime.chanrecv` function!

If we were actually trying to find primes, this would be a huge miss. But it is a great opportunity for us to show how micro-optimization can completely change the texture of your hot loop. Let's see if we can fix this up so we spend most of our time on the 'real work'.

Caching - that is, reusing intermediate results - is one of the core techniques of good performance. The result of `ctx.Done()` doesn't vary from call to call, so let's move that out of the loop and see if it changes our performance profile.

We move `context.Done()` to it's own variable right before the `SPIN` label, then run the profiler again...

`pprof`: cpu: examining 'optimized' `ewaste`:

```
#!/usr/bin/env bash
$ go tool pprof --seconds 5 http://localhost:6060/debug/pprof/profile?seconds=5
Fetching profile over HTTP from http://localhost:6060/debug/pprof/profile?seconds=5
Please wait... (5s)
Saved profile in /home/efron/pprof/pprof.ewaste.samples.cpu.006.pb.gz
File: ewaste
Build ID: 77b2195bda33dbcdc6681d905d56b0dd1bff4142
Type: cpu
Time: 2026-03-03 16:49:26 PST
Duration: 5s, Total samples = 14.94s (298.60%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 14920ms, 99.87% of 14940ms total
Dropped 1 node (cum <= 74.70ms)
      flat  flat%   sum%        cum   cum%
   12920ms 86.48% 86.48%    14940ms   100%  main.wasteCPU
    1010ms  6.76% 93.24%     1570ms 10.51%  runtime.chanrecv
     560ms  3.75% 96.99%      560ms  3.75%  runtime.empty
     430ms  2.88% 99.87%     2000ms 13.39%  runtime.selectnbrecv

(pprof) list waste
Total: 14.94s
```

```
ROUTINE ======================== main.wasteCPU in /home/efron/go/src/gitlab.com/efronlicht/ewaste/main.go
     12.92s     14.94s (flat, cum)   100% of Total
         .          .    192:func wasteCPU(ctx context.Context) []int {
         .          .    193:   primes := append(make([]int, 0, 4096), 2, 3, 5, 7, 11, 13, 17, 19, 23)
         .          .    194:   done := ctx.Done() // this used to be the most expensive thing in the program and now its so cheap as to be invisible.
         .          .    195:SPIN:
         .          .    196:
         .          .    197:   for {
         .          .    198:           select {
      600ms      2.60s    199:           case <-done: // probably this is runtime.chanrecv
```

```
11          70ms       70ms     200:                    return
    primes
12             .          .     201:             default:
13          300ms      300ms     202:                    for i :=
    primes[len(primes)-1]; i < math.MaxInt32; i += 2 {
14          490ms      490ms     203:
    for _, p := range primes[1:] {
15         11.18s     11.20s
    204:                                    if i%p == 0
    { // not prime
16          30ms       30ms
    205:                                            continue
    SPIN
17             .          .
    206:                                    }
18             .          .     207:                        }
19          250ms      250ms     208:
    primes = append(primes, i)
20             .          .     209:                    }
21             .          .     210:
22             .          .     211:              }
23             .          .     212:    }
24             .          .     213:}
```

Great! Now the cost breakdown looks like this:

- ~75% actually testing primes
- ~17% checking for cancellation

Not bad for a single line change. On to memory.

pprof: heap

Let's take a look at memory usage. The **heap profile** collects a statistical sample of memory operations throughout the entire program's operation, so it can be retrieved immediately.

Heap profiles work almost exactly like cpu profiles.

pprof: heap: collecting a profile

`go tool pprof localhost:6060/debug/pprof/heap` will retrieve the sample.

```
1  #!/usr/bin/env bash
2  go tool pprof localhost:6060/debug/pprof/heap
3  # <omitted>
4  Entering interactive mode (type "help" for commands, "o"
   for options)
```

Just like before, we can use `top` and `list` to find the hot spots.

sh 2: pprof: examine heap with `top` (2)

```
(pprof) top
Showing nodes accounting for 17694.61kB, 99.81% of
17728.53kB total
Dropped 207 nodes (cum <= 88.64kB)
      flat  flat%   sum%        cum   cum%
 9000.02kB 50.77% 50.77%  9000.02kB 50.77%  main.wasteMem
    8600kB 48.51% 99.28%     8600kB 48.51%  main.wasteCPU
   94.15kB  0.53% 99.81%    94.15kB  0.53%
runtime.mallocgc
    0.44kB 0.0025% 99.81%  9000.69kB 50.77%  main.main
         0     0% 99.81%  9002.44kB 50.78%  runtime.main

```

Let's dig a little deeper in and see where our memory is actually being used. We can use `list` and a regexp to find the actual *code* where our memory is being used.

Both of those functions contained the word `waste`, so let's start with that...

> Note: this `list` is from a different run, so the numbers aren't going to match the above. Don't worry about it.

sh 2: pprof: examine heap with `list` and regexp

```
$ (pprof) list waste
     8.79MB     8.79MB (flat, cum)  7.35% of Total
          .          .    314:func wasteMem(n int) {
     8.79MB     8.79MB    315:    m := make([]byte, n)
          .          .    316:    rng.Read(m)
        24B        24B    317:    wastebuf =
append(wastebuf, m)
          .          .    318:    wasted.mem.allocs++
          .          .    319:    wasted.mem.bytes += n
          .          .    320:    io.Discard.Write(m)
          .          .    321:}
```

It's really that easy. Don't guess where your performance problems are, *measure and find out*.

In this case, of course, there's nothing to optimize - these functions are *designed* to waste resources. But armed with this knowledge, it's time to look back at `cableshow`.

`pprof`: other profiles

While the CPU and Heap (memory) profiles are by far the most common and useful profilers, they're not the only ones. The documentation for `runtime/`

`pprof.Profile` is the place to learn more. > I'll reproduce part of it's summary here:

- `goroutine` - stack traces of all current goroutines
- `goroutineleak` - stack traces of all leaked goroutines
- `allocs` - a sampling of all past memory allocations
- `heap` - a sampling of memory allocations of live objects
- `threadcreate` - stack traces that led to the creation of new OS threads
- `block` - stack traces that led to blocking on synchronization primitives
- `mutex` - stack traces of holders of contended mutexes

Practical Optimization

(or: or how I saved >$300k/y with a handful of lines)

I told you that story so I could tell you this one: **how I optimized `cableshow` to save the ISP $300k/y**.

optimizing cableshow: profiling

I hooked up the heap profiler to a few of our production `cableshow` instances to figure out what was using all that memory.

It looked something like this:

- 63GiB - 70% of memory used on `formatOID(int) string` and `parseOID(string) int`
- 13.5GiB used for SNMP network traffic
- 13.5GiB for everything else.
- TOTAL: 90GiB steady state, spikes of up to 130GiB

  This kind of split is not unusual at all when profiling a program - hot spots are the rule, not the exception.

This is actually what you *want* to find. If your performance problems are spread out without any obvious locus, they're likely at a higher architectural level and much harder to fix. Clearly, performance improvements in `formatOID` and `parseOID` would pay off - especially `formatOID`.

`cableshow`: overview: Object Identifiers (OIDs)

Let's quickly review OIDs.

> An object identifier is a series of period-separated decimal numbers representing a hierarchical system, not dissimilar to an IPv4 address, but an OID is not limited to four sections of 0-255.
>
> To quote Harald T Alvestrand:
>
> > Object identifiers are, basically, strings of numbers. They are allocated in a hierarchical manner, so that, for instance, the authority for "1.2.3" is the only one that can say what "1.2.3.4" means.

Within the ISP, OIDs were at most five sections of at most three digits each - that is, the largest possible OID was `999.999.999.999.999`. 999,999,999,999,999 ~= 2^50, which means it easily fits in an int64. Some of the ISP's services treated the OID as a string, and some treated it as an `int64`. Internally, we treated it as an `int64`, so that `999.000.999.999.999` and `999.0.999.999.999` would be treated the same, but our SNMP client and some of our downstream customers wanted it as a string. This meant we were doing multiple conversions per cable modem per customer.

Let's take a quick look at the original code before we jump straight to solutions.

`cableshow.formatOID`: first attempts: `FormatOIDFoolish` and `FormatOIDNaive`

We'd always had the idea that this code would be at least a *little* performance sensitive. The very first pass at this code never made it past review for that reason - there's *some* level of 'defensive' performance programming you should always do. I'll present both and their benchmarks here:

```
 1
 2   const N_SECTIONS = 5
 3   const MUL = 1000
 4   const MAXOID int64 = (MUL * MUL * MUL * MUL * MUL) - 1 //
     the maximum OID we can represent with this format, which is
     999.999.999.999.999.
 5   const maxStrLen = len(`999.999.999.999.999`)           //
     five sections of three digits plus four dots.
 6
 7   // the original version of this code, rejected at code
     review.
 8   // this is the kind of code you should never let in your
     codebase.
 9   // in fact, the modern go analyzers will automatically warn
     you about things like this.
10   // https://pkg.go.dev/golang.org/x/tools/go/analysis/
     passes/modernize#hdr-Analyzer_stringsbuilder
11   func formatOIDFoolish(n int64) string {
12       var dst string
13       for {
```

```
14              section := n % MUL
15              n /= MUL
16              dst += strconv.FormatInt(section, 10) // this
    allocates two new strings on the heap for each section: one
    for the formatted section and one for the new dst string.
17              if n == 0 {
18                  break
19              }
20              dst += "." // this also allocates a new string on
    the heap.
21          }
22          return dst
23      }
24
25      // a sensible, straightforward conversion of an integer to
    an OID string.
26      // this was the original implementation that passed review
    and perfectly fine code.
27      func formatOIDNaive(n int64) string {
28          var buf bytes.Buffer
29          for {
30              section := n % MUL
31              n /= MUL
32              formatted := strconv.FormatInt(section, 10) //
    usually allocates.
33              buf.WriteString(formatted)                  //
    allocates if the buffer needs to grow, which will happen at
    least once per OID and probably log2(MUL) times per OID on
    average.
34              if n == 0 {
35                  break
36              }
37              buf.WriteByte('.') // more to go, so write a dot.
    may allocate if the buffer needs to grow.
38          }
39          return buf.String() // always allocates.
40      }
```

`cableshow.formatOID`: **Benchmarking** `FormatOIDNaive` **and** `FormatOIDFoolish`

Benchmarks exist to compare solutions for a known performance bottleneck. Now that we have a bottleneck - `formatOID` - now's the time to write a benchmark.

The basic form of a go benchmark is simple:

```
1   func myfunc() {/* code goes here*/}
2   func BenchmarkMyFunc(b *testing.B) {
3       for b.Loop() {
4           myfunc()
5       }
6   }
```

Benchmarks are *so* simple that it can be easy to write bad ones. For example, if we only ever benchmark a specific OID,
we learn the performance of our algorithm *for that specific one*, not OIDs in general. It is a good idea to have randomized input to benchmarks that approximates realistic input data. Ideally, you should benchmark both against synthetic data and against real samples the kind of data you want to process in production.

Here, we just generate random OIDs:

```
1   // generate random OIDs in the range [0, 9999999999999999)
    as integers.
2   func randOIDs(n int64) []int64 {
3       oids := make([]int64, n)
4       for i := range oids {
5           oids[i] = int64(rand.IntN(999_999_999_999_999 + 1))
6       }
7       return oids
8   }
9   var o =
    randOIDs(1000) // both of our benchmarks should use the
    same OIDs
10  func BenchmarkFormatNaive(b *testing.B) {
11
12      // try to do as little work that isn't the function
    you're trying to test in the benchmark.
13      // it would be inappropriate to call randOIDs INSIDE
    the loop.
14      for i := 0; b.Loop(); i++ {
15          j := i % len(o)
16          _ = formatOIDNaive(o[j])
17      }
18  }
19  func BenchmarkFormatOIDFoolish(b *testing.B) {
20      for i := 0; b.Loop(); i++ {
21          j := i % len(o)
22          _ = formatOIDFoolish(o[j])
23      }
24  }
```

INPUT:

```
1   #!/usr/bin/env bash
2
3   # run benchmarks matching the pattern 'Format'
4   $ go test -bench Format -benchmem
```

OUTPUT:

```
1   goos: linux
2   goarch: amd64
3   pkg: gitlab.com/efronlicht/blog/articles/startingsystems/
    bench/oid
4   cpu: AMD Ryzen 9 9900X 12-Core Processor
```

```
5   BenchmarkFormatOIDFoolish-24
    7814990               153.8 ns/op           119 B/
    op          12 allocs/op
6   BenchmarkFormatNaive-24         11659200             98.08 ns/
    op          102 B/op           6 allocs/op
7   PASS
8   ok    gitlab.com/efronlicht/blog/articles/startingsystems/
    bench/oid    2.336s
```

The CPU is not very taxed here - anything under 100ns is quite good - but six allocations per format and one per parse certainly can add up.

`cableshow.formatOID`: benchmarks: problems

- Try optimizing `formatOID` and `parseOID` to reduce the total number of allocations. Does knowing the maximum possible length of the OID help you?
- Examine the generated assembly of your solution with `go tool objdump`. See my article `Go Quirks & Tricks 3` for an example.
- Generate a tool to format benchmark output into nicer tables that don't repeat B/op, etc. over and over again, like the one in this article. Compare your solution to `gitlab.com/efronlicht/fmtbench`.

`cableshow.formatOID`: optimization #1: pre-allocation and `strings.Builder`

`FormatOID` seems like the obvious place to start, since it's doing far more allocations. Two small optimizations should help a lot:

- use `strings.Builder` instead of `bytes.Buffer` to save one allocation per OID
- pre-grow the buffer to the maximum possible length of an OID to save multiple allocations per OID.

`cableshow.formatOID`: #1: pre-allocation: code

```
1   // use strings.Builder instead of bytes.Buffer, which saves
    at least one allocation converting from bytes to OID
2   // use an intermediate stack-allocated buffer to format the
    integer before writing to destination
3   // to avoid heap-allocating intermediate strings
4   func formatOIDStringBuilder(n int64) string {
5       var dst strings.Builder // use a strings.Builder
    instead of bytes.Buffer, which is slightly more efficient
    for building strings.
6       dst.Grow(maxStrLen + 1) // pre-allocate the maximum
    length of the OID string to avoid multiple allocations as
    we build it.
7
```

```
        for {
            section := n % MUL
            n /= MUL
            dst.WriteString(strconv.FormatInt(section, 10))
            if n == 0 {
                break
            }
            dst.WriteByte('.') // more to go, so write a dot.
    this may allocate if the buffer needs to grow, but it won't
    allocate for the section.
        }
        return dst.String() // allocates a new string on the
    heap.
    }

    // benchmark in oid_bench_test.go...

    func BenchmarkFormatOIDStringBuilder(b *testing.B) {
        o := randOIDs(1000)
        for i := 0; b.Loop(); i++ {
            j := i % len(o)
            _ = formatOIDStringBuilder(o[j])
        }
    }
```

[cableshow.formatOID: #1: pre-allocation: benchmarks]

> From now on, I'm going to format the benchmarks as a table using [gitlab.com/efronlicht/fmtbench]. Long-time readers may recognize an older version of this program from my first-ever article, but this one's brand new.

IN

```
#!/usr/bin/env bash
go install gitlab.com/efronlicht/fmtbench@latest
go test -bench -benchmem Format | fmtbench
```

OUT

name	count	ns/op	B/op	allocs/op
BenchmarkFormatNaive-24	11812275	97.50	102	6
BenchmarkFormatOIDFoolish-24	7815867	154.1	119	12
BenchmarkFormatOIDStringBuilder-24	15118926	78.21	38	5

Estimated memory use: `63 GiB * (38/102) ~= 23.4 GiB`

cableshow.formatOID: #1: pre-allocation: profile

This helps a lot, cutting nearly two-thirds of the bytes under the average case, which by itself might be enough to let us use 128GiB machines and save some cash. Let's see if we can do better, though.

We can use the `-memprofile` and `-memprofilerate` flags to get a heap profile of our benchmarks to analyze later with `pprof`.

> ### profiling rates
>
> Keeping track of literally every operation or allocation is pretty expensive, so most profilers are "sampling" profilers and only track a percentage. You can adjust the profiling rate to 1 if you want to catch everything - that's what `ewaste` did for clarity. (After all, it wasn't trying to be efficient).

IN

```
#!/usr/bin/env bash

# write a pprof-readable profile to 'mem.out', and sample every single allocation.
# the actual benchmark times will be completely wrong because profililing is expensive, so throw them away.
$ go test -bench FormatOIDStringBuilder -memprofile mem.out -memprofilerate 1 2>&1 > /dev/null

# open an interactive pprof session, just like we did with `ewaste`.
$ go tool pprof mem.out
(pprof) top
```

OUT

```
15834.16kB 62.22% 62.22% 15834.16kB 62.22%  strings.(*Builder).grow
9507.03kB 37.36% 99.58%  9507.03kB 37.36%  strconv.formatBits
0.48kB 0.0019% 99.58% 25360.02kB 99.65%  testing.(*B).runN
0     0% 99.58% 25349.20kB 99.61%  gitlab.com/efronlicht/blog/articles/startingsystems/bench/oid.BenchmarkFormatOIDStringBuilder
0     0% 99.58% 25341.20kB 99.58%  gitlab.com/efronlicht/blog/articles/startingsystems/bench/oid.formatOIDStringBuilder
0     0% 99.58%  9507.03kB 37.36%  strconv.FormatInt
0     0% 99.58% 15834.16kB 62.22%  strings.(*Builder).Grow
0     0% 99.58% 25350.25kB 99.61%  testing.(*B).run1.func
```

IN

```
(pprof) list FormatInt
```

OUT

```
ROUTINE ======================== strconv.FormatInt in /usr/local/go/src/strconv/itoa.go
         0     9.28MB (flat, cum) 37.36% of Total
         .          .     25:func FormatInt(i int64, base int) string {
         .          .     26:   if fastSmalls && 0 <= i && i < nSmalls && base == 10 {
         .          .     27:           return small(int(i))
         .          .     28:   }
         .     9.28MB     29:   _, s := formatBits(nil, uint64(i), base, i < 0, false) // <-- HERE!
         .          .     30:   return s
         .          .     31:}
```

It's the allocation in `formatBits` that causes the problem.

`cableshow.formatOID: #2: FormatInt -> AppendInt`

We're doing a little too much work: rather than just formatting the bits, we're formatting the bits *and then allocating an intermediate string* (via `strconv.FormatInt`). We could instead format into a preallocated fixed-size buffer that's large enough to hold any section, and then copy *that* into the `strings.Builder`. We can do this using `strconv.AppendInt`, which is just like `strconv.FormatInt`, but you can pass it a buffer to skip allocations.

In fact, if you look at the source code, it's almost identical, but we pass in a buffer...

```
func AppendInt(dst []byte, i int64, base int) []byte {
    u := uint64(i)
    if i < 0 {
        dst = append(dst, '-')
        u = -u
    }
    if base == 10 {
        if i < nSmalls {
            return append(dst, small(int(i))...)
        }
        var a [24]byte
        j := formatBase10(a[:], i)
        return append(dst, a[j:]...)
    }
    dst, _ = formatBits(dst, i, base, false, true)
}
```

Once more into the breach:

FormatOIDAppendInt: Source

```
1   func formatOIDAppendInt(n int64) string {
2       dst := make([]byte, 0, maxStrLen+1) // pre-allocate a
    byte slice to build the OID string, which will avoid
    multiple allocations as we build it.
3       for {
4           section := n % MUL
5           n /= MUL
6           dst = strconv.AppendInt(dst, section, 10)
7           if n == 0 {
8               break
9           }
10          dst = append(dst, '.') // more to go, so write a
    dot. this may allocate if the buffer needs to grow, but it
    won't allocate for the section.
11      }
12      return string(dst) // allocates a new string on the
    heap.
13  }
14
15  func BenchmarkFormatOIDAppendInt(b *testing.B) {
16      o := randOIDs(1000)
17      for i := 0; b.Loop(); i++ {
18          j := i % len(o)
19          _ = formatOIDAppendInt(o[j])
20      }
21  }
```

cableshow.formatOID: #2: FormatInt -> AppendInt: benchmarks

IN

```
1   #!/usr/bin/env bash
2   go test -bench Format -benchmem ./startingsystems/bench/oid
```

OUT

name	count	ns/op	B/op	allocs/op
BenchmarkFormatNaive-24	12303301	96.18	102	6
BenchmarkFormatOIDAppendInt-24	32016249	38.02	23	1
BenchmarkFormatOIDStringBuilder-24	14865608	79.25	38	5

This looks great. We're down to a single allocation and an estimated memory use for string formatting of around `16GiB (63 GiB) * (23 / 102) ~= 14.2 GiB`).

Looks great. We're running in a third the time and a fifth the memory.

`cableshow.FormatOID #2: Problems for the Reader

- `strings.Builder` is more efficient than `bytes.Buffer`. Why? Examine the source code to find out. Is this a trick you should use? Why or why not?
-

`cableshow.formatOID`: #3: string tables

It *seems* like we're operating close to our theoretical maximum: 38 nanoseconds is very fast, and for randomly distributed strings, you can't do better than one allocation per string on average. For a *general-purpose* conversion algorithm - like one you'd put in an `oid` library - you'd probably want to stop here. In fact, when I actually *had* this job and did this optimization, I *did* stop here, because I couldn't think of any way to do better. But I'm a much better system programmer now than I was back then, and it turns out there's still plenty of room for improvement.

Pushing allocations to the back of our mind for now, let's take another look at the CPU profile to see if we can find some further opportunities for savings.

`cableshow.formatOID`: #3: examining the profile of optimization #2

IN

```
#!/usr/bin/env bash
go test -cpuprofile=cpu.out -bench AppendInt -benchmem .
2>&1 > /dev/null
go tool pprof cpu.out
(pprof) top
```

OUT

```
Showing top 10 nodes out of 59
      flat  flat%   sum%        cum   cum%
     360ms 28.35% 28.35%      360ms 28.35%  runtime.memmove
     320ms 25.20% 53.54%      630ms 49.61%
strconv.formatBits
     170ms 13.39% 66.93%     1180ms 92.91%  gitlab.com/
efronlicht/blog/articles/startingsystems/bench/
oid.formatOIDAppendInt
```

```
 6        110ms  8.66% 75.59%      750ms 59.06%
    strconv.AppendInt
 7         80ms  6.30% 81.89%      150ms 11.81%
    runtime.mallocgcSmallNoscan
 8         60ms  4.72% 86.61%       60ms  4.72%  runtime.futex
 9         40ms  3.15% 89.76%      190ms 14.96%
    runtime.mallocgc
10         20ms  1.57% 91.34%       20ms  1.57%
    runtime.nextFreeFast (inline)
11         20ms  1.57% 92.91%       20ms  1.57%
    runtime.publicationBarrier
12         20ms  1.57% 94.49%      260ms 20.47%
    runtime.slicebytetostring
```

Well, probably not much we can do about `runtime.memmove`, but is there some way we can avoid `formatBits` and `AppendInt`?

Here's where knowledge of our problem domain kicks in. Unlike a general purpose OID algorithm, we know that each subsection of *our* OIDs are between `0` and `999`, with a `.` at the end for every one but the last one. This brings two tricks to mind:

- If we had a table of those small integer strings preallocated, we could just look them up via without needing to do the math in `strconv`.
- Strictly speaking, an OID has more sections that end with a `.` than those that don't - only the last one doesn't. If we store the strings with the `.` at the end, we'll do less work most of the time.

`cableshow.formatOID`: #3: string tables: generation with `genoidtable`

Let's **write a program** to generate the oid table: `genoidtable`

```
 1  // genoidtable.go generates a table with every possible OID
    subsection
 2  package main
 3
 4  import (
 5      "fmt"
 6  )
 7
 8  func main() {
 9      fmt.Print(`
10  const formatTable = "`)
11      for i := range 1000 {
12
13          fmt.Printf("%3d.", i) // pre-format the section
    with a dot, padding with spaces to ensure consistent length
    so we can slice directly into it.
14          if i%100 == 0 && i != 0 {
```

```
15                fmt.Println(`"+`) // add a newline every 100
    sections so badly programmed IDEs like VSCode don't choke
    on them.
16                fmt.Print(`"`)
17            }
18        }
19        fmt.Println(`"`)
20
21    }
```

Which gives an output that looks like this:

cableshow.formatOID: #3: string tables: generated output

```
1   package oid
2
3   // generated by ./cmd/genoidconv, which is a simple program
    that generates a table of pre-formatted sections of the OID.
    this allows us to convert an integer back to an OID string
    with only a single allocation, which is the final string.
    the conversion code can slice directly into this table to
    get the pre-formatted section, which is much faster than
    using strconv.FormatInt for each section.
4   const formatTable = "  0.  1.  2.  3.  4.  5.  6.  7.  8.
    9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23.
    24. 25. 26. 27. " /* many omitted */
5   + /* many omitted*/ ".992.993.994.995.996.997.998.999."
```

The stdlib does this for two-digit integers - it's where I learned the trick.

This certainly costs us some ram - slightly under 4KiB, but with this this table in our pocket, we can write an even *faster* version of formatOID:

```
1   // conversion of an integer back to an OID string with only
    a single allocation and with extremely fast lookup by
2   // pre-calculating a table with all possible subsections of
    the OID. this allows us to convert an integer back to an
    OID string with only a single allocation, which is the
    final string. the conversion code can slice directly into
    this table to get the pre-formatted section, which is much
    faster than using strconv.FormatInt for each section.
3   // see cmd/genoidconv for the code that generates the
    formatTable constant.
4   func formatOIDStringTable(n int64) string {
5       var buf [maxStrLen + 1]byte // one extra for trailing
    '.' which we'll strip off.
6       var i int
7       for range N_SECTIONS {
8           section := n % MUL
9           off := 4 * section // each section has four bytes
    reserved for it in the conversion table, so we can
    calculate the offset easily.
```

```
10          // we do need to shift a little bit to account for
    the fact that the sections are not all the same length.
11
12          var formatted string
13          switch {
14          case section >= 100: // `ddd.`
15              formatted = formatTable[off : off+4]
16          case section >= 10: // ` dd.`
17              formatted = formatTable[off+1 : off+4]
18          default: // `  d.`
19              formatted = formatTable[off+2 : off+4]
20          }
21          i += copy(buf[i:], formatted) // write the
    formatted section into the buffer. this will not heap-
    allocate because we already pre-allocated the buffer.
22
23          n /= MUL // shift down to the next section.
24          if n == 0 {
25              return string(buf[:i-1]) // we parsed the last
    section. strip off the trailing dot and return the result.
26          }
27      }
28      panic(fmt.Sprintf("too many sections: expected at most
    %d, got more than that", N_SECTIONS)) // this should be
    impossible because we only have N_SECTIONS iterations, but
    we'll panic just in case.
29  }
```

[cableshow.formatOID: #3: string tables: benchmarks]

IN

```
1  #!/usr/bin/env bash
2  go test -bench Format -benchmem .
```

OUT

name	count	ns/op	B/op	allocs/op
BenchmarkFormatNaive-24	12160269	96.79	102	6
BenchmarkFormatOIDAppendInt-24	31243256	38.14	23	1
BenchmarkFormatOIDStringBuilder-24	14995860	79.24	38	5
BenchmarkStringTableFormatOID-24	50805165	23.94	23	1

IN

```
1  #!/usr/bin/env bash
2
```

```
# run the profile...
go test -cpuprofile=cpu.out -bench StringTable -benchmem . 2>&1 > /dev/null
go tool pprof cpu.out
(pprof) top
```

OUT

```
      flat  flat%   sum%        cum   cum%
     450ms 37.50% 37.50%     1020ms 85.00%  gitlab.com/efronlicht/blog/articles/startingsystems/bench/oid.formatOIDStringTable
     170ms 14.17% 51.67%      170ms 14.17%  runtime.memmove
     100ms  8.33% 60.00%      100ms  8.33%  runtime.nextFreeFast (inline)
      70ms  5.83% 65.83%       70ms  5.83%  runtime.futex
      60ms  5.00% 70.83%      290ms 24.17%  runtime.mallocgcSmallNoscan
```

This cuts another 40% off our runtime. Let's take one last look at the CPU profile with `list` to see if there's anything else we can do.

IN

```
(pprof) list formatOID
```

```
Total: 1.20s
ROUTINE ======================== gitlab.com/efronlicht/blog/articles/startingsystems/bench/oid.formatOIDStringTable in /home/efron/go/src/gitlab.com/efronlicht/blog/articles/startingsystems/bench/oid/oid.go
     450ms      1.02s (flat, cum) 85.00% of Total
         .          .    127:func formatOIDStringTable(n int64) string {
         .          .    128:   var buf [maxStrLen + 1]byte
         .          .    129:   var i int
      20ms       20ms    130:   for range N_SECTIONS {
     170ms      170ms    131:           section := n % MUL
      10ms       10ms    132:           off := 4 * section // each section has four bytes reserved for it in the conversion table, so we can calculate the offset easily.
         .          .    133:           // we do need to shift a little bit to account for the fact that the sections are not all the same length.
         .          .    134:
         .          .    135:           var formatted string
         .          .    136:           switch {
         .          .    137:           case section >= 100: // `ddd.`
```

```
15          60ms       60ms    138:                    formatted
    = formatTable[off : off+4]
16          30ms       30ms    139:            case section >=
    10: // ` dd.`
17          10ms       10ms    140:                    formatted
    = formatTable[off+1 : off+4]
18             .          .    141:            default: // `  d.`
19             .          .    142:                    formatted
    = formatTable[off+2 : off+4]
20             .          .    143:            }
21       130ms      220ms    144:            i += copy(buf[i:],
    formatted) // write the formatted section into the buffer.
    this will not heap-allocate because we already pre-
    allocated the buffer.
22             .          .    145:
23             .          .    146:            n /= MUL // shift
    down to the next section.
24          20ms       20ms    147:            if n == 0 {
25             .      480ms    148:                    return
    string(buf[:i-1]) // we parsed the last section. strip off
    the trailing dot and return the result.
26             .          .    149:            }
27             .          .    150:    }
28             .          .    151:
    panic(fmt.Sprintf("too many sections: expected at most %d,
    got more than that", N_SECTIONS)) // this should be
    impossible because we only have N_SECTIONS iterations, but
    we'll panic just in case.
29             .          .    152:
30             .          .    153:}
```

This is pretty close to the studs.Looks like we're spending our time doing four things: allocating the final string (480ms), copying the sections (220ms), and taking the modulus (the % operator, 170ms). I don't think we can optimize this much further without platform-specific tricks.

It still feels like we could avoid some allocations, though - let's get back to that.

`cableshow.FormatOID`: #3: Problems

> Line 146 is 'free', even though we know division is expensive. Why? Try looking at the generated assembly to find out.

```
1   >>          .          .    146:
    n /= MUL // shift down to the next
    section.
2   >> ```
3
4   ### `cableshow.formatOID` #4: Interning
    and Memoization
5
```

```
6   Is there anything we know about our
    problem domain that would let us omit some
    allocations? Well, for one thing, there
    are 'only' about thirty million cable
    modems, and they are partitioned in a
    relatively consistent way across
    instances, so each instance will only see
    a few million of them at most. Maybe it
    would be cheaper to allocate each OID as
    it came up _exactly once_ and then just
    hold it in memory for the rest of the
    lifecycle of the program? This would
    remove garbage collection of those strings
    entirely, at the cost of higher RAM usage
    when there isn't much load.
7
8   Let's do some back of the envelope math to
    see whether this is a sane idea.
9
10  #### Optimization 4: Back-of-the-envelope
    math
11
12  We have between 1 and 30 million OIDs in
    our dataset. We know these OIDs have
    string representations less than 20
    characters long.
13  We can store these as unique interned
    values, using the number as the key in a
    `sync.Map` and a `unique.Handle[string]`
    as the value.
14  Let's make a rough estimate of how much
    this will cost:
15
16  - 8 bytes per key (the int64)
17  - Add 24+16+8 bytes per OID (rounding up
    to nearest multiple of word size for the
    string, then adding 16 bytes for the
    string header (ptr + len) and 8 for the
    unique.Handle overhead...)
18  - ... double that because hash tables
    aren't densely packed
19  - ... and multiply by 1.5x just for
    overhead (you always undercount)
20  - which ends up as (56 _ 2) _ 1.5 = 168
    bytes per OID
21  - multiply by 30 million...
22  - **5,040,000,000 bytes, or a little under
    5GiB of RAM, even if we want to hold every
    single expected OID in memory at the same
    time.**
23
```

```
This could be dangerous if we're suddenly confronted with millions of new OIDs, but this condition is not likely to suddenly change, because installing millions of new cable modems is

- something the ISP would know about ahead of time
- a physical task that involves drilling holes, wiring, etc.

#### Optimization 4: `FormatOIDMemoized`: Code

```go
// lookup the OID in the memoization cache, using the cached result if possible.
// if no cached result is found, convert the integer to an OID string using the optimized method and update the cache with the new OID and integer.
// this is safe for concurrent use.
// the underlying function will take the maps as arguments so we can benchmark it without using globals,
// but the 'real' implementation will hide the details from our callers, either by providing a Cache type or by using package-level variables.
func formatOIDMemoized(int2OID *sync.Map, n int64) string {
    v, ok := int2OID.Load(n) // has someone else already formatted this?
    if ok {                  // yes.
        return v.(unique.Handle[string]).Value() // return the interned string from the cache.
    }
    s := formatOIDStringTable(n)              // format in the most efficient way we have.
    interned := unique.Make(s)                // intern the string so we only store one copy of it in memory.
    got, _ := int2OID.LoadOrStore(n, interned) // store that interned string.
    return got.(unique.Handle[string]).Value() // return the interned string from the cache, which may be the one we just stored or one that another goroutine stored at the same time.
}
```
```

Let's take a peek at the benchmarks...

```
func BenchmarkMemoizedFormatOID(b *testing.B) {
    o := randOIDs(1000)
    var int2OID sync.Map
    for i := 0; b.Loop(); i++ {
        j := i % len(o)
        _ = formatOIDMemoized(&int2OID, o[j])
    }

}
```

Optimization 4: `FormatOIDMemorized: Benchmarks`

IN

```
#!/usr/bin/env bash
go test -bench Format -benchmem .
```

OUT

name	count	ns/op	B/op	allocs/op
BenchmarkFormatNaive-24	12167324	97.99	102	6
BenchmarkFormatOIDAppendInt-24	31266152	38.94	23	1
BenchmarkFormatOIDFoolish-24	7603273	154.9	119	12
BenchmarkFormatOIDStringBuilder-24	15368214	78.12	38	5
BenchmarkMemoizedFormatOID-24	85718442	12.68	0	0
BenchmarkStringTableFormatOID-24	48893323	24.62	23	1

Well, that seems about as good as we're going to get. A couple caveats to keep in mind: formatting isn't actually "free" - we still allocate on the heap, it's just that as we repeatedly hit the cache over and over, the *average case allocation* approaches zero. We still have plenty of stuff in the heap - and it will stay there effectively 'forever', even if all possible garbage is collected - but once everything is allocated, we will never need to do it again.

`cableshow.formatOID`: Optimization #4: Interning and Memoization: Advanced Memory Metrics

The above is probably plenty good reasoning, but if that wasn't good enough for us and we wanted to find out *exactly* how much memory was being kept alive by our cache, we could use `runtime.ReadMemStats` or the `metrics` package to find out.

cableshow.formatOID: #4: problems: “custom metrics done badly”

I got the code for the following section wrong repeatedly. The following code contains at least three errors at two different conceptual levels. What are they, and why?

```
>> // call this AFTER any setup and BEFORE b.Loop() to report on allocations and heap growth during operation.
>> // ... is what this function is supposed to do.
>> func reportSteadyStateHeapGrowthWRONG(b *testing.B) { //
>>     tb.Helper()
>>     var before runtime.MemStats
>>     runtime.ReadMemStats(&before) // read the mem stats
>>
>>     tb.Cleanup(func() { // like 'defer' for test scope
>>         var after runtime.MemStats
>>         runtime.ReadMemStats(&after)
>>
b.ReportMetric(float64(after.Mallocs - before.Mallocs), "allocs")
>>
b.ReportMetric(float64(after.HeapAlloc - before.HeapAlloc/1024/1024), "MiB_heap_growth")
>>     })
>> }
>> ```
>> ##### Answers: Custom Metrics Done Badly
>> - Division (3x error)
>>   - cast to float64 happens after subtraction, which can cause unsigned integer wraparound
>>   - cast to float64 happens _before_ division, truncating!
>> - grouping is wrong: should be (`after.HeapAlloc`-`before.HeapAlloc`).
>> - no guarantee that `before` doesn't unrelated garbage from previous functions
>> - no guarantee that `after` hasn't already cleaned up benchmark variables</details>

The following code is extremely subtle and can be considered 'advanced material'.

```

```go
// report heap stats for a benchmark.
// call this immediately before b.Loop().
// types passed in `keepalive` must be pointer-like (interface, map, slice, chan, func, or pointer, or string)
func reportHeapStats(b *testing.B, keepalive ...any) {
    b.Helper()
    var stats runtime.MemStats
    // add a metric to the benchmark with label label_mib
    heap := func(label string) float64 {
        const MiB = 1024 * 1024
        runtime.ReadMemStats(&stats)
        x := float64(stats.HeapAlloc) / MiB
        b.ReportMetric(x, label+"_mib")
        return x
    }
    for _, v := range keepalive {
        switch v := reflect.TypeOf(v); v.Kind() {
        // homework problem: why are these kinds of types "pointer-like"? what is the runtime representation of each type?
        case reflect.Interface, reflect.Map, reflect.Slice, reflect.Chan, reflect.Func, reflect.Ptr, reflect.String:
            // nop
        default:

panic(fmt.Errorf("reportSteadyStateHeapGrowth: passed a value type (%s, a %s) to runtime.KeepAlive. Nothing to keep alive: this is a value type", v, v.Kind()))
        }
    }

    debug.SetGCPercent(-1) // turn off automatic GC completely for the duration of the benchmark
    runtime.GC()           // cleanup before we read the initial memory stats
    before := heap("before")

    b.Cleanup(func() {
        b.Helper()
        hot := heap("hot")      // heap holding EVERYTHING allocated since we turned the gc off.

debug.SetGCPercent(100) // restore ordinary GC behavior.
```

```
60            runtime.GC()              // cleanup
61            cold := heap("cold")      // heap
    after reclaiming everything possible (not
    including 'nogc')
62            b.ReportMetric(cold-hot,
    "reclaimed_mib")
63            b.ReportMetric(cold-before,
    "leaked_mib")
64
65            for _, v := range keepalive {
66                runtime.KeepAlive(v) // tell
    the compiler (and therefore runtime) we
    still need this value until AFTER the call
    to runtime.GC() above.
67            }
68
69        })
70    }
```

Using advanced memory metrics: final benchmarks

We'll add that in to each of our benchmarks

```
1   func BenchmarkMemoizedFormatOID(b *testing.B) {
2       o := randOIDs(1000)
3       var int2OID sync.Map
4       reportHeapStats(b, &int2OID) // keep int2OID alive
5       for i := 0; b.Loop(); i++ {
6           _ = formatOIDMemoized(&int2OID, o[i%len(o)])
7       }
8   }
```

And run the benchmark one last time, just to make sure we haven't missed anything:

IN

```
1   #!/usr/bin/env bash
2   go test -bench Format -benchmem ./startingsystems/bench/oid
```

OUT

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_
BenchmarkFormatNaive-24	10417634	150.0	0.3286	1017	-0.002174	-1016
BenchmarkFormatOIDAppendInt-24	31170796	42.42	0.3289	709.0	-0.007286	-708.7
BenchmarkFormatOIDFoolish-24	6862900	178.1	0.3290	781.7	-0.007286	-781.4
BenchmarkFormatOIDStringBuilder-24	14894722	85.06	0.3288	545.6	-0.007164	-545.2

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_
BenchmarkMemoizedFormatOID-24	84051058	12.77	0.6559	0.6867	0.3145	-0.03082
BenchmarkStringTableFormatOID-24	47326560	28.43	0.3341	1073	-0.002281	-1072

We've now proved our point - while this is *much, much, much* more efficient than any of the other solutions in terms of GC churn, it costs us about 317 bytes per ID (`317 , 600 / 1000`) of 'permanent' heap storage. **It's worth noting that this is double what our back-of-the-envelope math predicted, despite our generous fudge factors, but we were in the right ballpark.**

This is even better than it appears, because **we no longer have spikes under high demand**, which means we don't have to pay for headroom to protect ourselves.

Let's take a look at `parseOID` before we move on to other targets.

Optimizing `cableshow.parseOID`

There's a lot less room for improvement here, so I'll skip straight to showing you the naive and optimized code and explain why it's faster in the comments.

Optimizing `cableshow.parseOID`: code

```
// parse an OID of up to MAX_SECTIONS into an integer in a straightforward way using strings.Split and strconv.Atoi. this is not super efficient but it's perfectly reasonable programming.
func parseOIDNaive(s string) (int64, error) {
    m := strings.Split(s, ".") // this allocates a slice of strings on the heap, which is always a little expensive.
    if len(m) > N_SECTIONS {
        return 0, fmt.Errorf("too many sections: expected at most %d, got %d", N_SECTIONS, len(m))
    }

    var result int64
    // parse each section, treating the right hand side as the least significant.
    for i := len(m) - 1; i >= 0; i-- {
        n, err := strconv.Atoi(m[i])
        if err != nil {
            return 0, fmt.Errorf("invalid section %d: %w", i, err)
        }
        if n < 0 || n >= MUL { // each section must be between 0 and MUL, exclusive.
```

```
16                return 0, fmt.Errorf("section %d out of range: expected 0 <= n < %d , got %d", i, MUL, n)
17            }
18            result = result*MUL + int64(n)
19        }
20        return result, nil
21    }
22
23    // parse OIDs with no allocations at all.
24    func parseOIDNoAlloc(s string) (int64, error) {
25        var result int64
26        var nsection int
27
28        // rather than using 'strings.Split', we'll manually split it up starting from the back, skipping the slice allocations.
29        for ; nsection < N_SECTIONS; nsection++ {
30            i := strings.LastIndexByte(s, '.')
31            // s[i+1]: is the section we want to parse, not including the dot. if i == -1, this will be s[0], which is correct because we want to parse the whole string.
32            n, err := strconv.ParseInt(s[i+1:], 10, 64) // parse the section as an integer. if this fails, we can return an error immediately.
33            if err != nil {
34                return 0, fmt.Errorf("invalid section: %w", err)
35            }
36
37            if n < 0 || n >= MUL {
38                return 0, fmt.Errorf("section out of range: expected 0 <= n < %d , got %d", MUL, n)
39            }
40            result = result*MUL + int64(n)
41            if i == -1 {
42                return result, nil // we parsed the last section, so we can return the result.
43            }
44            s = s[:i] // remove the parsed section and the dot.
45        }
46        return 0, fmt.Errorf("too many sections: expected at most %d, got at least %d", N_SECTIONS, nsection+1)
47    }
```

Let’s run the benchmarks…

`cableshow.parseOID`**: benchmarks**

IN

```
1   #!/usr/bin/env bash
```

```
go test -bench Parse -benchmem ./startingsystems/bench/oid
```

OUT

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_mib	B/op
BenchmarkParseNaive-24	29114545	51.66	0.3339	1340	-0.02577	-1340	48
BenchmarkParseOIDNaive-24	23391009	58.48	0.3285	1092	-0.02571	-1091	48
BenchmarkParseOIDNoAlloc-24	43778109	27.46	0.3288	0.3602	-0.03065	-0.03144	0

`ParseOIDNoAlloc` **is twice as fast and uses no heap memory at all. Seems pretty solved to me.**

SNMP: Evaluation after optimization of `cableshow.parseOID` and `cableshow.formatOID`

We pushed our new versions to production and started collecting heap profiles on our new, improved program.

SNMP: Next Steps

Once you've solved a bottleneck, take a step back and re-run your profilers and metrics before you move on to the next thing.

- ~13.5 GiB on SNMP networking
- ~10GiB on OIDs (down from 70GIB)
- 13.5 GiB on 'everything else'.
- TOTAL: 37 GiB (down from 90GiB - saved 60%)

Our clear bottleneck was now the SNMP networking code. A glance at `pprof` told us that the allocations were happening during calls to `NewConn`:

> Note: the specific data here and library used are simplified to make teaching easier.

SNMP: example code (large buffer)

```
package examplesnmp
type Conn struct {
    conn net.Conn // underling network connection
    buf *[0xFFFF]byte // <- allocated for the lifetime of the connection
    /* other fields omitted*/
}
```

```
 7  func NewConn(ctx context.Context, oid int) (*Conn, error) {
 8      buf := new([0xFF]byte) // <---- HERE!
 9      /* much code omitted*/
10
11      c := &Conn{
12          buf: buf
13           /* much code omitted*/
14      }
15      return c, nil
16  }
17
18  func (c *Conn) ReadPacket() ([]byte, error) {n, err :=
    c.conn.Read(c.buf[:]); return c.buf[:n], err}
19
```

This was for a simple and sensible reason: SNMP packets were sent over IP/UDP. UDP is stateless: if you miss a packet, it's gone. Even though **most** of the packets our modems sent us were a few hundreds bytes, if they *did* send us one that was larger than our buffer, we'd miss it. **The only way to guarantee you catch every packet is to use a buffer at least as large as the largest possible packet - 64kib**.

SNMP: simplifying the code before evaluation

To make it easier to benchmark, let's get rid of the structs here and just deal with the buffer directly. Instead of reading from the network, we'll just read from anything, too.

```
1  // read using a preallocated fixed-size buffer, like the
   original code.
2  func FixedBuffer(b *[0xFFFF]byte, read func(b []byte) (n
   int, err error)) ([]byte, error) {
3      n, err :=
   read(b[:]) // no allocation, but we hold on to the buffer
   forever.
4      if err != nil {
5          return nil, err
6      }
7      return b[:n], nil // no allocation, but we hold on to
   the buffer forever.
8  }
```

Allocating a buffer for the lifetime of the collection means you can't miss and you create no garbage, at the cost of 65k of ballast per conn. For the vast majority of uses, this is perfectly fine - you can have 16 buffers per MB, which means 10k connections will cost you under a GiB. For example, a World of Warcraft server's 4000 simultaneous connections would use only 250MiB. (Not that WoW uses SNMP, but just as an example). Unfortunately, *we* didn't have ten thousand simultaneous connections - we had hundreds of thousands.

SNMP: “Optimization” 1: A truly terrible ‘solution’

My first proposed solution was also straightforward: allocate a big buffer for the read, then copy the data to a smaller buffer once you knew how big the packet was. Then the garbage collector could reclaim the big buffer.

```
1  // read allocating fresh every time.
2  func AlwaysAllocate(read func(b []byte) (n int, err error))
   ([]byte, error) {
3      b := make([]byte, 0xFFFF) // large allocation!
4      n, err := read(b)
5      if err != nil {
6          return nil, err
7      }
8      return bytes.Clone(b[:n]), nil // small allocation!
9  }
```

It turned out *reads* are much more common than *connections*, and allocating a buffer per *read* is far worse. So much worse it crashed our staging servers. Whoops.

SNMP: 2 `simple.Pool`

Even though we can have tens of thousands of simultaneous network connections, even the most powerful computers can’t actually read ten thousand packets simultaneously: they have a limited number of cores. In theory, then, we didn’t need to allocate a big buffer *per connection*, we needed to allocate a buffer *per simultaneous read*.

One solution to this problem would be allocating buffer *per thread*, but Go doesn’t have thread-local variables (and they come with a whole host of problems of their own).

A handy data structure for this kind of thing is a *shared pool*. When you want a buffer, check whether there’s one in the shared pool first and take that if you can. If you can’t, make a new buffer - but *either way, return that buffer to the pool when you’re done*. This will grow the pool to a steady size that matches your peak simultaneous connections. Under relatively consistent demand the number of allocations should approach zero, at the cost of some ‘steady-state’ wasted memory. (The parallels to the previous optimizations using `sync.Map` should stick out).

Go contains a very good pool - `sync.Pool` - but let’s write our own first to get the idea.

SNMP: `simple.Pool`

```
1  package readsnmp
2
3  import (
4      "sync"
5  )
6
```

```
// Simple, type-specific shared pool of buffers using a mutex and slice. Prefer `sync.Pool` to this in real code.
// Safety: do NOT call `Put` on a buffer that is still in use: this is racy.
//
// Usage:
//
//  var sp SimplePool
//  buf := sp.Get() // get a buffer, possibly making a new one.
//  /* use buf */
//  sp.Put(buf) // return the buffer to the pool for later use.
type SimplePool struct {
    mux                sync.Mutex // guards stack
    a                  []*[0xFFFF]byte // stack of buffers
}

// Get a buffer, or create a new one if no buffer exists.
func (sp *SimplePool) Get() *[0xFFFF]byte {
    sp.mux.Lock()
    defer sp.mux.Unlock()
    if len(sp.a) == 0 { // nothing in the pool. make a fresh buffer and hand it out.
        return new([0xFFFF]byte)
    }
    // pop the last buffer off the stack.
    i := len(sp.a) - 1
    buf := sp.a[i]
    sp.a = sp.a[:i]
    return buf
}

// Return the buffer to the pool for later use.
func (sp *SimplePool) Put(b *[0xFFFF]byte) {sp.mux.Lock(); sp.a = append(sp.a, b); sp.mux.Unlock()}


// read using a shared pool of 'large' buffers guarded by mutex, copying the result into a newly allocated small buffer.
func SharedSimplePool(sp *SimplePool, read Read) ([]byte, error) {
    b := sp.Get()
    defer sp.Put(b)
    n, err := read(b[:])
    if err != nil {
        return nil, err
    }
    return bytes.Clone(b[:n]), nil
}
```

SNMP: 3 `sync.Pool`.

Let's also compare it to go's `sync.Pool.

```
1   // read using a shared 'large' buffer via *sync.Pool,
    copying the result into a newly allocated small buffer.
2   func SyncPoolAlwaysAlloc(sp *sync.Pool, read Read)
    ([]byte, error) {
3       b := sp.Get().(*[0xFFFF]byte)
4       defer sp.Put(b)
5       n, err := read(b[:])
6       if err != nil {
7           return nil, err
8       }
9       return bytes.Clone(b[:n]), nil
10  }
```

SNMP: Evaluation Strategies

Let's compare the following strategies:

1. The original fixed buffer
2. Always allocating
3. `SimplePool`
4. `sync.Pool`

SNMP: Benchmarking and Evaluation

Just like before, we want to have make sure our benchmarks are somewhat realistic. Reads over network connections in our system are

- of wildly varying sizes
- have significant latency
- are extremely concurrent - it's not unusual to have hundreds of thousands of connections!

If our benchmarks simulate different conditions, they might give us the right answer to the wrong question!

That means running the benchmarks in parallel on a **distribution** of data.

SNMP: Benchmarking: Running Parallel Benchmarks

Running parallel benchmarks is easy:

```
1   func myFunc() {/* code goes here */}
2
```

```
func BenchmarkMyFuncParallel(b *testing.B) {
    b.RunParallel(func(p *testing.PB) {
        for p.Next() {
            myFunc()
        }
    })
}
```

SNMP: Building Data Distributions

Let's **write some utility functions** to try and build better benchmarks. Ideally, we'd have benchmarks that included samples captured from real data to best match our problem domain, but we can at least build a variety of synthetic packets that are somewhat realistic.

```
// --- helper functions and vars to represent slightly more realistic packets ---
var (
    packets   [][]byte        = buildPackets(100_000)
    latencies []time.Duration = buildLatencies(100_000)
)

func buildPackets(n int) [][]byte {
    packets := make([][]byte, n)
    for i := range packets {
        // build a representative packet between 0xFF and 0xFFFF bytes
        // centered around 400 bytes.
        const mean, stddev = 400, 200
        size := int(rand.NormFloat64()*stddev + mean) // avg: mean, normally distributed with stddev
        size = max(0xFF, min(0xFFFF, size)) // clamp 0xFF<=size<=0xFFFF
        b := make([]byte, size)
        b[rand.IntN(size)] = byte(rand.IntN(256)) // set a random byte to make sure the compiler doesn't optimize out making the slice
        packets[i] = b
    }
    return packets
}

// build a slice of n latencies exponentially distributed around a mean of 5ms
func buildLatencies(n int) []time.Duration {
    latencies := make([]time.Duration, n)
    for i := range latencies {
        // latency is extremely variable. let's use an exponential distribution with a mean of 5ms to model it.
        // to quote the documentation of rand.ExpFloat64:
        d := time.Duration(rand.ExpFloat64() / float64(5*time.Millisecond))
```

```
29          // exponential distributions can go all over the
    place, so let's put a cap on them.
30          d = max(time.Millisecond, d)
31          d = min(time.Second, d)
32          latencies[i] = d
33
34      }
35      return latencies
36  }
37
38  // fakeUDPRead simulates a packet read into 'b'. assume a
    latency evenly distributed between 1ms and 50ms
39  func fakeUDPRead(b []byte) (n int, err error) {
40      i := rand.IntN(len(packets))
41      delay, packet := latencies[i], packets[i]
42      time.Sleep(delay)
43
44      n = copy(b, packet)
45      if n < len(packet) {
46          return n, io.ErrShortBuffer
47      }
48      return n, nil
49
50  }
```

SNMP: benchmark code

Let's put it all together and write our benchmarks.

```
 1  func BenchmarkFixedLargeBuffer(b *testing.B) {
 2      // vary the # of conns to see how the different
    strategies perform under different levels of contention.
 3      for conns := 100; conns <= 100_000; conns *= 10 {
 4
 5          b.Run(fmt.Sprintf("%010d", conns), func(b
    *testing.B) {
 6              b.SetParallelism(conns) // tell the benchmark
    to use up to `conns` simultaneous goroutines during
    b.RunParallel.
 7              buf := make([][0xFFFF]byte, conns)
 8
 9              var i atomic.Int64
10              i.Store(-1)
11              b.ResetTimer()
12              reportHeapStats(b, &buf)
13
14              b.RunParallel(func(p *testing.PB) {
15                  b := &buf[i.Add(1)%int64(len(buf))]
16                  for p.Next() {
17                      readsnmp.FixedBuffer(b, fakeUDPRead)
18                  }
19              })
```

```
            })
        }
    }

    func BenchmarkAlwaysAllocate(b *testing.B) {

        for conns := 100; conns <= 10_000; conns *= 10 { /
    *100_000 crashed my computer, so we do slightly less ;) */
            b.Run(fmt.Sprint(conns), func(b *testing.B) {
                reportHeapStats(b)
                b.SetParallelism(conns)
                b.ResetTimer()
                b.RunParallel(func(p *testing.PB) {
                    for p.Next() {
                        readsnmp.AlwaysAllocate(fakeUDPRead)
                    }
                })
            })
        }
    }
    func BenchmarkSyncPoolAlwaysAlloc(b *testing.B) {
        for conns := 100; conns <= 100_000; conns *= 10 {
            b.Run(fmt.Sprint(conns), func(b *testing.B) {
                b.SetParallelism(conns)
                pool := sync.Pool{New: func() any { return
    new([0xFFFF]byte) }}
                reportHeapStats(b, &pool)
                b.RunParallel(func(p *testing.PB) {
                    for p.Next() {
                        readsnmp.SyncPoolAlwaysAlloc(&pool,
    fakeUDPRead)
                    }
                })
            })
        }
    }
    func BenchmarkSyncPoolFixedSmall(b *testing.B) {
        for conns := 100; conns <= 100_000; conns *= 10 {
            b.Run(fmt.Sprint(conns), func(b *testing.B) {
                b.SetParallelism(conns)
                pool := sync.Pool{New: func() any { return
    new([0xFFFF]byte) }}
                small := make([][0x400]byte, conns)
                var i atomic.Int64
                i.Store(-1)
                reportHeapStats(b, &pool, &small)
                b.RunParallel(func(p *testing.PB) {
                    smallbuf := &small[i.Add(1)
    %int64(len(small))]
                    for p.Next() {
                        readsnmp.SharedLargeFixedSmall(&pool,
    smallbuf, fakeUDPRead)
                    }
                })
```

```
68
69              })
70          }
71      }
72
73      func BenchmarkSimplePool(b *testing.B) {
74
75          for conns := 100; conns <= 1_000_000; conns *= 10 {
76              b.Run(fmt.Sprint(conns), func(b *testing.B) {
77                  b.SetParallelism(conns)
78                  var sp readsnmp.SimplePool
79                  reportHeapStats(b, &sp)
80
81                  b.RunParallel(func(p *testing.PB) {
82                      for p.Next() {
83                          readsnmp.SharedSimplePool(&sp,
    fakeUDPRead)
84                      }
85                  })
86              })
87          }
88      }
```

SNMP: benchmark results

IN

```
1   #!/usr/bin/env bash
2   go test -bench . -benchmem . ./bench/readsnmp
```

Out:

Fixed Large Buffer (original)

The fixed large buffer generates very little garbage but uses too much memory at steady-state.

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_mib	B/op
BenchmarkFixedLargeBuffer/0000000100-24	1683372	707.2	59.68	53.46	60.17	-6.220	-6.714
BenchmarkFixedLargeBuffer/0000001000-24	9437870	125.6	177.8	115.3	180.7	-62.46	-65.44
BenchmarkFixedLargeBuffer/0000010000-24	9441368	124.7	1303	678.5	1323	-624.8	-644.4

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_mib	B/op
BenchmarkFixedLargeBuffer/0000100000-24	9127796	129.8	12554	6304	12738	-6250	-6434

Always Allocate

Always allocating is 10x slower and uses more memory under nearly every circumstance. The 'cure' was worse than the disease.

name	count	ns/op	cold_mib	hot_mib	leaked_mib	reclaimed_mib	B/op	allocs/op
BenchmarkAlwaysAllocate/100-24	759810	1518	53.73	376.2	-0.005775	-322.5	445	1
BenchmarkAlwaysAllocate/1000-24	694106	1638	53.73	350.2	-0.005775	-296.5	447	1
BenchmarkAlwaysAllocate/10000-24	395817	2662	53.73	240.4	-0.005867	-186.7	494	2

note the missing benchmark for 100k here: this crashed my computer.

`SimplePool` benchmarks`

Simplepool performs well under moderate concurrency, saving a lot of 'cold' (steady-state) memory at the expense of being slower than the fixed large buffer. Unfortunately, this doesn't scale well past 10000 or so connections, **since everything's waiting on acquiring a single lock!**

name	count	ns/op	before_mib	cold_mib	hot_mib	leaked_mib	reclaime
BenchmarkSimplePool/100-24	1823428	644.9	205.1	205.0	1129	-0.02213	-923.7
BenchmarkSimplePool/1000-24	1420484	749.9	762.4	764.8	2079	2.457	-1314
BenchmarkSimplePool/10000-24	1682150	680.5	767.4	729.7	2176	-37.73	-1446
BenchmarkSimplePool/100000-24	1356668	756.1	727.1	713.1	2145	-14.04	-1432
BenchmarkSimplePool/1000000-24	1	3552107187	55.03	55.10	1886	0.06305	-1831

`sync.Pool` benchmarks

Unsurprisingly, the stdlib's `sync.Pool` performs the best of all our options.

The stdlib's `sync.Pool` solves this problem using sophisticated engineering beyond the scope of this article - that's *advanced* systems programming. [Take a look at the source for `sync.Pool` if you want to see how the magic happens.]

name	count	ns/op	before_mib	cold_mib	hot_mib	leaked_mib	reclai
BenchmarkSyncPoolAlwaysAlloc/100-24	1591567	771.1	53.74	203.8	878.9	150.1	-675.1
BenchmarkSyncPoolAlwaysAlloc/1000-24	8760234	116.6	54.76	805.1	4521	750.4	-3716
BenchmarkSyncPoolAlwaysAlloc/10000-24	9728583	117.0	54.76	805.8	4948	751.0	-4142
BenchmarkSyncPoolAlwaysAlloc/100000-24	9636267	112.1	54.77	805.1	5072	750.4	-4267

SNMP: Bonus Optimization: Small fixed buffer, shared large buffer

So we're done, right? `sync.Pool` is as good as it gets? Well, maybe. Let's think about our problem statement again.

While we must have a large buffer to catch the occasional large packet, most of this time is overkill: **most of our packets are small** - well under 1K. We could mirror the problem in our solution by giving each connection a small, permanent buffer for average-case reads, borrowing from the `sync.Pool` only long enough to 'catch' the UDP packet. (For unusually large packets, we can allocate - the garbage collector will do fine for occasional exceptions).

Since our average packet is about 400B, 1KiB should be more than enough to handle the vast majority of cases.

SNMP:

```
1   // read using the *sync.Pool's big buffer, copying it into
    `small` if the packet is small enough to fit,
2   // and allocating a new buffer otherwise.
3   func SharedLargeFixedSmall(sp *sync.Pool, small
    *[0x400]byte, read Read) ([]byte, error) {
4       b := sp.Get().(*[0xFFFF]byte) // possible allocation:
    under steady state, allocations approach 0.
5       defer sp.Put(b)
6       n, err := read(b[:])
7       if err != nil {
8           return nil, err
9       }
10      if n < len(small)
    { // this is an ordinary packet. copy into small.
```

```
            _ = copy(small[:], b[:n]) // no allocation, but a little bit of copying
            return small[:n], nil
        }
        // 'large' packet. copy it into a new buffer, and return that.
        return bytes.Clone(b[:n]), nil // moderate allocation.
    }

    func BenchmarkSyncPoolFixedSmall(b *testing.B) {
        for conns := 100; conns <= 100_000; conns *= 10 {
            b.Run(fmt.Sprint(conns), func(b *testing.B) {
                b.SetParallelism(conns)
                pool := sync.Pool{New: func() any { return new([0xFFFF]byte) }}
                small := make([][0x400]byte, conns)
                var i atomic.Int64
                i.Store(-1)
                reportHeapStats(b, &pool, &small)
                b.RunParallel(func(p *testing.PB) {
                    smallbuf := &small[i.Add(1)%int64(len(small))]
                    for p.Next() {
                        readsnmp.SharedLargeFixedSmall(&pool, smallbuf, fakeUDPRead)
                    }
                })

            })
        }
    }
```

Let's run the benchmarks again, just comparing our `sync.Pool` implementations...

IN

```
#!/usr/bin/env bash
go test -bench SyncPool -benchmem ./bench/readsnmp
```

OUT

name	count	ns/op	before_mib	cold_mib	hot_mib	leaked_mib	reclai
BenchmarkSyncPoolAlwaysAlloc/100-24	1439539	803.2	47.58	197.8	808.9	150.2	-611.1
BenchmarkSyncPoolAlwaysAlloc/1000-24	7669573	132.8	53.73	804.3	4060	750.6	-3256
BenchmarkSyncPoolAlwaysAlloc/10000-24	9495912	119.9	54.19	804.8	4851	750.6	-4046

name	count	ns/op	before_mib	cold_mib	hot_mib	leaked_mib	reclai
BenchmarkSyncPoolAlwaysAlloc/100000-24	8037630	124.9	55.12	806.4	4398	751.3	-3592
BenchmarkSyncPoolFixedSmall/100-24	1652472	735.4	55.32	205.3	207.5	150.0	-2.227
BenchmarkSyncPoolFixedSmall/1000-24	8933874	127.5	57.08	807.0	820.3	749.9	-13.27
BenchmarkSyncPoolFixedSmall/10000-24	9068928	125.9	74.76	816.1	854.6	741.3	-38.59
BenchmarkSyncPoolFixedSmall/100000-24	8920342	124.9	250.5	903.6	1195	653.0	-291.0

`cableshow`: final results

Let's take one last look at our three 'good' options under heavy load:

name	count	ns/op	before_mib	cold_mib	hot_mib	leaked_mib	reclai
BenchmarkFixedLargeBuffer/0000100000-24	9127796	129.8	12554	6304	12738	-6250	-6434
BenchmarkSyncPoolAlwaysAlloc/100000-24	8037630	124.9	55.12	806.4	4398	751.3	-3592
BenchmarkSyncPoolFixedSmall/100000-24	8920342	124.9	250.5	903.6	1195	653.0	-291.0

`FixedSmall` uses 15% of the steady-state, and runs about 4% faster (though this is probably within the margin of error), and has no spikes under demand, so long as large packets are rare. **Now that's optimization**.

Average Memory Use:

- ~2 GiB on SNMP networking (down from 15GiB)
- ~10 GiB on OIDs (down from 70GIB)
- ~13.5 GiB on 'everything else'.
- TOTAL (steady state): 25.5 GiB (down from 90GiB - **saved 72%**)
- TOTAL (peak): 28GiB (down from 130GiB - **saved 78%**)

This means we can go down from 256GiB machines to 32GiB machines, which are much, much, much, cheaper - about 10% of the price on average.

`cableshow`: final results: Problems

> `syncPoolAlwaysAlloc` uses more 'cold' memory than `SimplePool`. Why?

Conclusion

Long story short: **performance optimization pays**. My story of saving $300k/year here is striking but not extreme. I know engineers who have been able to save millions with similar techniques.

This article is easily longer than anything else than I've ever written and it still feels like I'm only scratching the surface - I barely talked about memory or hardware, had to leave out large sections about metrics and I/O, and couldn't even manage to sneak in a line of assembler. Still, I think we covered a lot of ground. This was great fun to write and I hope you enjoyed reading it. If you've made it this far, you've got the makings of a real systems programmer. Hold your head up proud and remember: **performance patters** - don't let anyone tell you it doesn't.

Bonus Material

`swtalmud`

Part of the reason why this article has taken so long is I've been hard at work at a very personal project, also taking the form of an article on this website. It's called "Software Talmud: The Anatomy of Automation" and I'd really appreciate it if you checked it out.

Heap Escapes, Memory Allocation, Benchmarks

The following are the scraps of a segment on memory allocations and heap escapes. I couldn't quite get it to come together as a section on it's own, but I think it's pretty illustrative as code, so I cleaned up the comments. A run of the benchmarks follows, followed by the code.

```
#!/usr/bin/env bash
go test -bench . -trimpath -benchmem ./articles/
startingsystems/bench/escape
```

```
#!/usr/bin/env bash
cat ./articles/startingsystems/bench/escape/escape_test.go
```

Heap Escapes: Benchmarking Code

```
package escape

import (
    "log"
    "runtime"
```

```
 6      "strconv"
 7      "strings"
 8      "testing"
 9  )
10
11  func init() {
12      log.Print(`// when benchmarking, it's important to
    measure the right thing.
13  // try and eliminate as many confounding factors as
    possible.
14  // for each benchmark, we'll allocate a big slab of memory
    up front before we start tracking.`)
15  }
16
17  func initbench(b *testing.B, comment string) (buf []byte)
    {
18      b.Logf("\n%s\n\t%s\n", b.Name(),
    strings.ReplaceAll(comment, "\n", "\n\t"))
19      b.Helper()
20      b.ReportAllocs()
21      buf = make([]byte, 0, GiB)
22      b.ResetTimer()
23      return buf
24  }
25
26  func BenchmarkAppendIntBaseline(b *testing.B) {
27      dst := initbench(b, `// where possible, always
    benchmark in comparison to a reference implementation.
28  // here, we'll use ordinary `+"`strconv.AppendInt`")
29      for n := 0; b.Loop(); n++ {
30          dst = strconv.AppendInt(dst, int64(n), 10)
31      }
32  }
33
34  func BenchmarkAppendBuf(b *testing.B) {
35      dst := initbench(b, `// let's build a deliberately
    less efficient version to get some ideas about optimizing
    memory usage.
36  // in this version, we'll first allocate the formatted
    integer in it's own buffer, then append it to our
    destination.
37  // go's escape analyzer is good enough to realize that we
    don't actually use the intermediate buffer, so it never
    escapes at all.`)
38
39      for n := 0; b.Loop(); n++ {
40          // allocate an intermediate buffer and stick the
    integer in it
41          p := strconv.AppendInt(nil, int64(n), 10)
42
43          // now append that to dst.
44          dst = append(dst, p...)
45
```

```
46          // at this point, p will no longer be touched by anything else in the program,
47          // so the compiler can avoid most heap allocations here by reusing the same space for the contents of p.
48      }
49  }
50
51  func BenchmarkAppendBufLeak(b *testing.B) {
52      dst := initbench(b, `
53      // if for some reason p DID live outside this loop, that optimization would no longer be valid.
54      // we can artificially provoke this with`+"`runtime.KeepAlive`"+`.
55      // let's build a deliberately less efficient version to get some ideas about optimizing memory usage.
56      // in this version, we'll first allocate the formatted integer in it's own buffer, then append it to our destination.
57      // go's escape analyzer is good enough to realize that we don't actually use the intermediate buffer, so it never escapes at all.
58  `)
59      for n := 0; b.Loop(); n++ {
60          // allocate an intermediate buffer and stick the integer in it
61          p := strconv.AppendInt(nil, int64(n), 10)
62
63          // now append that to dst.
64          dst = append(dst, p...)
65
66          defer runtime.KeepAlive(p)
67
68      }
69      b.Log(`all versions of p are still 'alive', so the compiler had to allocate afresh for each.`)
70  }
71
72  func BenchmarkAppendPreAllocNoEscape(b *testing.B) {
73      dst := initbench(b, `// even if the compiler can re-use that space, it still has to allocate SOME space on the heap b/c it doesn't have an
74  // idea how big the result will be.
75  // We can help the compiler out by picking a reasonable number ahead of time
76  // Go benchmarks run at most 100000000 times, which is 9 digits.
77  // This implementation is nearly as fast as the original: memcopies are incredibly fast even when they're pointless.`)
78      for n := 0; b.Loop(); n++ {
79          p := strconv.AppendInt(make([]byte, 0, 10), int64(n), 10)
80          dst = append(dst, p...)
81      }
```

```
 82  }
 83
 84  func BenchmarkAppendPreAllocEscape(b *testing.B) {
 85      dst := initbench(b, `// On the other hand, if that
     memory escapes to the heap, our pre-allocation helps
     significantly less.`)
 86      for n := 0; b.Loop(); n++ {
 87          p := strconv.AppendInt(make([]byte, 0, 10),
     int64(n), 10)
 88          dst = append(dst, p...)
 89          defer runtime.KeepAlive(p)
 90      }
 91  }
 92
 93  func BenchmarkAppendIntStrNoEscape(b *testing.B) {
 94      dst := initbench(b, `Even allocating a bunch of small
     strings can be cleverly inlined in simple situations...`)
 95      for n := 0; b.Loop(); n++ {
 96          s := strconv.Itoa(n)
 97          dst = append(dst, s...)
 98      }
 99  }
100
101  func BenchmarkAppendIntStrEscape(b *testing.B) {
102      dst := initbench(b, "// but if those strings escape
     it's the worst we've seen yet")
103      for n := 0; b.Loop(); n++ {
104          s := strconv.Itoa(n)
105          dst = append(dst, s...)
106          defer runtime.KeepAlive(s)
107
108      }
109  }
110
111  func BenchmarkAppendIntNoInline(b *testing.B) {
112      dst := initbench(b, "// function calls take time.
     Individual calls are neglitgible, but big chains of
     indirections can add up.")
113
114      for n := 0; b.Loop(); n++ {
115          dst = noinlineappend(dst, int64(n), 10)
116      }
117  }
118
119  // the 'go:noinline' compiler directive SUGGESTS not
     inlining the code.
120  //
121  //go:noinline
122  func noinlineappend(dst []byte, n int64, base int) []byte
     {
123      return strconv.AppendInt(dst, n, base)
124  }
125
```

Heap Escapes: Benchmarking Results

```
2026/04/23 21:25:16 // when benchmarking, it's important to measure the right thing.
// try and eliminate as many confounding factors as possible.
// for each benchmark, we'll allocate a big slab of memory up front before we start tracking.
goos: darwin
goarch: arm64
pkg: gitlab.com/efronlicht/blog/articles/startingsystems/bench/escape
cpu: Apple M2
BenchmarkAppendIntBaseline-8
100000000               11.17 ns/op             0 B/op          0 allocs/op
--- BENCH: BenchmarkAppendIntBaseline-8
    escape_test.go:18:
        BenchmarkAppendIntBaseline
            // where possible, always benchmark in comparison to a reference implementation.
            // here, we'll use ordinary `strconv.AppendInt`
BenchmarkAppendBuf-8
56065813                23.24 ns/op             8 B/op          1 allocs/op
--- BENCH: BenchmarkAppendBuf-8
    escape_test.go:18:
        BenchmarkAppendBuf
            // let's build a deliberately less efficient version to get some ideas about optimizing memory usage.
            // in this version, we'll first allocate the formatted integer in it's own buffer, then append it to our destination.
            // go's escape analyzer is good enough to realize that we don't actually use the intermediate buffer, so it never escapes at all.
BenchmarkAppendBufLeak-8                14613769               127.3 ns/op           104 B/op          4 allocs/op
--- BENCH: BenchmarkAppendBufLeak-8
    escape_test.go:18:
        BenchmarkAppendBufLeak

                // if for some reason p DID live outside this loop, that optimization would no longer be valid.
                // we can artificially provoke this with`runtime.KeepAlive`.
                // let's build a deliberately less efficient version to get some ideas about optimizing memory usage.
                // in this version, we'll first allocate the formatted integer in it's own buffer, then append it to our destination.
```

```
30                  // go's escape analyzer is good enough to
    realize that we don't actually use the intermediate buffer,
    so it never escapes at all.
31
32      escape_test.go:69: all versions of p are still
    'alive', so the compiler had to allocate afresh for each.
33      ... [output truncated]
34  BenchmarkAppendPreAllocNoEscape-8    100000000
    10.60 ns/op         0 B/op         0 allocs/op
35  --- BENCH: BenchmarkAppendPreAllocNoEscape-8
36      escape_test.go:18:
37          BenchmarkAppendPreAllocNoEscape
38              // even if the compiler can re-use that space,
    it still has to allocate SOME space on the heap b/c it
    doesn't have an
39              // idea how big the result will be.
40              // We can help the compiler out by picking a
    reasonable number ahead of time
41              // Go benchmarks run at most 100000000 times,
    which is 9 digits.
42              // This implementation is nearly as fast as the
    original: memcopies are incredibly fast even when they're
    pointless.
43  BenchmarkAppendPreAllocEscape-8        14656249
    122.6 ns/op       111 B/op         3 allocs/op
44  --- BENCH: BenchmarkAppendPreAllocEscape-8
45      escape_test.go:18:
46          BenchmarkAppendPreAllocEscape
47              // On the other hand, if that memory escapes to
    the heap, our pre-allocation helps significantly less.
48  BenchmarkAppendIntStrNoEscape-8        65944182
    22.85 ns/op         7 B/op         0 allocs/op
49  --- BENCH: BenchmarkAppendIntStrNoEscape-8
50      escape_test.go:18:
51          BenchmarkAppendIntStrNoEscape
52              Even allocating a bunch of small strings can be
    cleverly inlined in simple situations...
53  BenchmarkAppendIntStrEscape-8        13357376
    119.7 ns/op        95 B/op         3 allocs/op
54  --- BENCH: BenchmarkAppendIntStrEscape-8
55      escape_test.go:18:
56          BenchmarkAppendIntStrEscape
57              // but if those strings escape it's the worst
    we've seen yet
58  BenchmarkAppendIntNoInline-8
    100000000            10.43 ns/op         0 B/op         0
    allocs/op
59  --- BENCH: BenchmarkAppendIntNoInline-8
60      escape_test.go:18:
61          BenchmarkAppendIntNoInline
62              // function calls take time. Individual calls
    are neglitgible, but big chains of indirections can add up.
63  PASS
```

```
64   ok      gitlab.com/efronlicht/blog/articles/
     startingsystems/bench/escape  14.518s
```

MORE ARTICLES