Adopting continuous-profiling:

Understand how your code utilizes cpu/memory

Introduction into continuous-profiling and how it can help you writing more efficient code

2023-02-05

FOSDEM 2023 - Monitoring and Observability devroom
Christian Simon
Software Engineer at Grafana Labs

Working on observability databases (Loki, Mimir, Phlare)
Observability

- Introspect your applications/infrastructure running in production
- Objective way of looking at state between teams
- Goals
  - Avoid negative user experience and ideally catch problems before they become user facing
  - Reduce the mean time to repair
  - Aid in root cause analysis
Observability using Logs

- No specific instrumentation or application changes need, as most applications already support logging in some form
- Challenges
  - Aggregations across many log lines can become quite expensive
  - Different log formats and sometimes it might be hard to correlate information
  - Signal to noise ratio can hide important information
Observability using Metrics

- Numerical data measured over a time
- Typical metrics measured for web services:
  - RED method (Rate, Error rate and Duration of requests)
- A lot of values can be stored and aggregated efficiently
- Challenges
  - Applications require instrumentation, so it is important to know beforehand what to measure
Observability using Traces

- Introspect how requests, which are dependent on each other are flowing through a distributed system.
- Go from metrics to traces using exemplars
- Challenges
  - Not all requests might be sampled
Let’s go through an example

- User raises a ticket because during check out they saw a timeout
- Looking up the trace ID in the logs reveals that tracing show the location service has been timing out
- Looking at the metrics for all location service replicas, we can see 5% of the requests time out
- Next steps
  - Scale up replicas for location service 🏆🏆🏆🏆
  - Optimise location service
Observability using Profiles

Profiling shows the resource usage of an application

- Profiling information serves to aid program optimization, and more specifically, performance engineering.
- Profiling can help reduce and understand workload cost (TCO), improve service latency and fixes applications problems (OOM)
- Multiple types of profiling data
  - Space (memory): How much memory my application uses or allocates? And where?
  - Time (complexity): The frequency and duration of function calls. Where is my application spending most of CPU time?
  - And more.... threads, synchronization....
What is measured in a Profile?

```go
package main

func main() {
    // work
doALot()
doLittle()
}

func prepare() {
    // work
}

func doALot() {
    prepare()
    // work
}

func doLittle() {
    prepare()
    // work
}```
What is measured in a profile? Time on CPU

Each measurement gets recorded on a stack-trace level

```go
package main

func main() {
    // spend 3 cpu cycles
doALot()
doLittle()
}
func prepare() {
    // spend 5 cpu cycles
}
func doALot() {
    prepare()
    // spend 20 cpu cycles
}
func doLittle() {
    prepare()
    // spend 5 cpu cycles
}
```
Visualization of Profiles (try it yourself: https://pprof.me/b9d077f)

TOP table

- **Flat**: Consumption by the function only
- **Cumulative**: Consumption by the function and its descendants
- **Sum%**: Based on the order of the table how much of the total measured consumption is covered by the row

```
package main

func main() {
    // spend 3 cpu cycles
    doALot()
    doLittle()
}

func prepare() {
    // spend 5 cpu cycles
}

func doALot() {
    prepare()
    // spend 20 cpu cycles
}

func doLittle() {
    prepare()
    // spend 5 cpu cycles
}
```

<table>
<thead>
<tr>
<th>Flat</th>
<th>Flat%</th>
<th>Sum%</th>
<th>Cum</th>
<th>Cum%</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>52.63%</td>
<td>52.63%</td>
<td>25</td>
<td>65.79%</td>
<td>doALot</td>
</tr>
<tr>
<td>10</td>
<td>26.32%</td>
<td>78.95%</td>
<td>10</td>
<td>26.32%</td>
<td>prepare</td>
</tr>
<tr>
<td>5</td>
<td>13.16%</td>
<td>92.11%</td>
<td>10</td>
<td>26.32%</td>
<td>doLittle</td>
</tr>
<tr>
<td>3</td>
<td>7.89%</td>
<td>100.00%</td>
<td>38</td>
<td>100.00%</td>
<td>main</td>
</tr>
</tbody>
</table>
Visualization of Profiles  (try it yourself: https://pprof.me/b9d077f)

Flamegraph

- Whole width represent the total resources used (over the whole measurement duration)
- Ability to spot higher usage nodes
- Colours are random

```
package main
func main() {
    // spend 3 cpu cycles
doALot()
doLittle()
}
func prepare() {
    // spend 5 cpu cycles
}
func doALot() {
    prepare()
    // spend 20 cpu cycles
}
func doLittle() {
    prepare()
    // spend 5 cpu cycles
}
```
How to gather a profile?

- Instrumenting the code base
  - Tooling and formats depending on each language ecosystem
  - Access to more detailed runtime information
- eBPF based collection
  - No insights into runtime information
    (so better suited for compiled languages)
  - Doesn’t require instrumentation of application
How to gather a profile? Let’s take a look at Go

- Standard library includes CPU, Memory, Goroutine, Mutex and Block resources
- Provides profiles using a HTTP interface
  - Profiling data is returned using protobuf definition
- Data meant to be consumed by the pprof CLI
  - # Get a CPU profile over the last 2 seconds
    
    $ pprof "http://localhost:6060/debug/pprof/profile?seconds=2"

  - # Get the heap memory allocations
    
    $ pprof "http://localhost:6060/debug/pprof/allocs"

  - Common to use the -http parameter to view profiles using the web interface
- Find more on Profiling in Go on https://pkg.go.dev/runtime/pprof#Profile
package main

import {
    "log"
    "net/http"
    "net/http/pprof"
    "time"
}

func main() {
    go func() {
        log.Println(http.ListenAndServe("localhost:6060", nil))
    }()
    // spend 3 cpu cycles
    doALot()
    doLittle()
}

[...]
The challenges with deterministic profiling

- Significant runtime overhead
- Hard to recreate problematic scenarios
- Even harder in distributed systems / microservices
- Large volume of profiling data
Continuous profiling

- Championed by Google in production
- Sampling the call stack
- Sampling $\Rightarrow$ very low overhead
- “Always on” in production
A typical continuous profiling workflow

- **Your application**
- **Store / Query / Visualize**
- **Optimize**

Flowchart details:
- Scrape profiles
- Iterate

Diagram elements:
- Your application -> Scrape profiles -> Store / Query / Visualize -> Iterate -> Optimize
Store / Query / Visualize

- pprof CLI and profile collection at scale can become tedious
- Multiple solutions exist
- Profiling databases can simplify the workflow
  - CNCF Pixie
  - Pyroscope
  - Polarsignal Parca
  - Grafana Phlare
Demo time 🌡️

Your application

Phlare
Storage + query

Grafana
Visualize

Optimize

iterate

scrape profiles

query
Profile guided optimizations

- “Optimize” step of the workflow typically involves a human reasoning about profiling data and the code.
- Compilers can also do Profile guided optimization (PGO).
- Having production/real world profiling information allows to improve decision making at compile time.
- Go 1.20 includes PGO in public review, which improves the inlining decision making.
https://github.com/simonswine/demo-proprof

#phlare on https://grafana.slack.com/

https://grafana.com/docs/phlare/latest/

https://play-phlare.grafana.org/