Monitoring Performance of Golang Applications Using Code Profiling

Nilesh Jagnik
Mountain View, USA
nileshjagnik@gmail.com

***Abstract*—Debugging and monitoring are essential for building scalable and reliable applications. Often, real-world applications have performance regressions. These can be hard to root cause using conventional debugging and monitoring tools. Code profiling tools can be quite handy for such scenarios. Profiling tools can be used to detect hotspots, i.e., parts of code that have poor performance and high resource utilization. Profiling tools provide deep insights into the runtime behavior of an application which can be useful in general. The pprof package can be used for profiling of Golang applications. This tool is easy to use and can provide information about a lot of runtime aspects of a program. In this paper, we take a look at pprof, how to use it in Golang and review the benefits and caveats associated with its use.**

Keywords—software monitoring, software performance, code profiling

I. INTRODUCTION

While building and maintaining software at large scale, eventually there may be some unexpected behavior in the system. This could be in the form of performance regressions, reduced throughput, increased latency, etc. The root cause for such issues could vary a lot. Some examples are memory leaks, expensive computations, programming errors or even external RPCs. Detecting unexpected behavior is especially important when applications are required to scale to large workloads. The problem is that often the root causes are hard to detect.

There are many debugging tools that developers can use to detect issues in code. These could be as simple as printing logs to get visibility into the internal state of the system. Or running the program in debug mode using a debugger to step through each line of code. These approaches can detect simple problems with code. But for real world applications, it is hard to form a bigger picture of an application's health using these approaches. There is a need to get a top-down view of the system to understand where the bottlenecks actually are.

One way to get a wholistic view of an application is to get the thread dump. This would help get a snapshot of the system to see its running state. However, a snapshot still captures one instant of time and may not be representative of what is actually happening over a longer period of time. The best way to get what we want is by using profiling tools.

Profiling tools give us an overview of different properties of a running a running program, which can be used to study the behavior of different parts of code during runtime. Profiling tools can provide visibility into an application's performance which is quite hard to get using any other tool. Golang has support for pprof, which is a profiling tool that can provide visual analysis of system data. This tool is quite easy to install and use and is almost indispensable for real-world Golang applications.

II. WHAT ARE PROFILING TOOLS

Profiling tools (or profilers) capture the running state of a program, including CPU, memory, and utilization of several other resources. In addition to utilization, they also capture the distribution of resources utilized by different parts of code. The data captured by profilers (called profiles) can usually be converted into text reports or visualized as graphs and figures for easier understanding. Profilers can capture data over a time interval to profile the average state of a program over time. All these qualities make profilers an excellent tool

for learning about nuanced behaviors exhibited by different parts of a program or server.

III. REASONS TO USE PROFILING TOOLS

A. *Identifying Performance Bottlenecks*

Profilers can be used to pinpoint parts of code that consume excessive resources, and cause performance regressions. Developers gain visibility into the problematic parts of the system. Development cycles can then be focused on improvements that will lead to the largest performance gains.

```
import (
  "fmt"
  "net/http"
  "sync"
  // This import is necessary
  _ "net/http/pprof"
)

func cpuIntensiveTask(wg *sync.WaitGroup) {
  defer wg.Done()
  for {
    acc := 1
    for i:= 0; i < 10000000; i++ {
      acc *= i;
    }
  }
}

func memoryIntensiveTask(wg *sync.WaitGroup) {
  defer wg.Done()
  data := make([]int, 1)
  for {
    data = append(data, 10000000)
  }
}

func main() {
  go func() {
    fmt.Println(http.ListenAndServe("localhost:6060",
                                    nil))
  }()
  var wg sync.WaitGroup
  wg.Add(1)
  go cpuIntensiveTask(&wg)
  wg.Add(1)
  go memoryIntensiveTask(&wg)
  wg.Wait()
}
```

B. *Insight into Application Behavior*

```
$ go tool pprof -web
http://localhost:6060/debug/pprof/profile?seconds=60
```

Profilers are not just useful when the performance of the system has deteriorated. Continuously profiling the system provides insight into how it behaves under different scenarios. Profiling is a great tool for asserting that the system behaves as expected even under load. This can help ensure scalability of services.

C. *Optimization of Resource Utilization*

When certain resources are expensive, it is important to write code that will optimally use those resources. Profiling can be used to detect when certain resources are leaking or being wasted. This can lead to reduced costs for running services.

D. *Enhancing User Experience*

Profiling ensures that a service has good performance. In addition to performance, it also ensures that services are stable and reliable. This leads to improved user experience.

E. *Preventing Waste of Development Time*

Making incremental updates to an application's code and profiling it ensures that development time is not wasted doing work which later needs to be reverted.

IV. PROFILING WITH PPROF

The pprof package in Golang can be used for easy creation of profiles. These profiles can be visually represented with the help of Graphviz. Let us take a look of how to use pprof.

A. *Code Setup*

The use of pprof requires some setup in code to allow the collection of profiles. The first thing that is needed is to import the pprof package into your program. Then, if the application isn't already running an HTTP server, it must be started to allow interactions with the profiler at runtime. Fig. 1 shows an example program that has some memory intensive and CPU intensive tasks. It imports the pprof package and runs an HTTP server. We will use run the profiler on this program.

Fig. 1. An example that uses pprof for profiling.

B. *CPU Profile*

Now that the code it set up, it is easy to collect a CPU profile. Fig. 2 shows the command that must be run from the terminal to look at a 60s CPU profile. The use of the -web flag tells the pprof to generate a graphical view.

Fig. 2. Command for generating a 60s CPU profile.

C. *Analyzing Profiles*

The command in Fig. 2 prompts the creation of a graph showing the CPU resource consumption distribution. Fig. 3 shows the graph generated for the

code in Fig. 1. We can see that even though the CPU intensive task consumes ~40% of CPU resources, there are two other tasks which are using the remaining resources. The first of these is the memory intensive task which uses up ~39% of CPU resources – almost as such the CPU intensive task. Apart from this, there is also a garbage collection task which uses 20% of CPU resources. This reveals to us that the memory intensive task is equally expensive CPU-wise as the CPU intensive task.

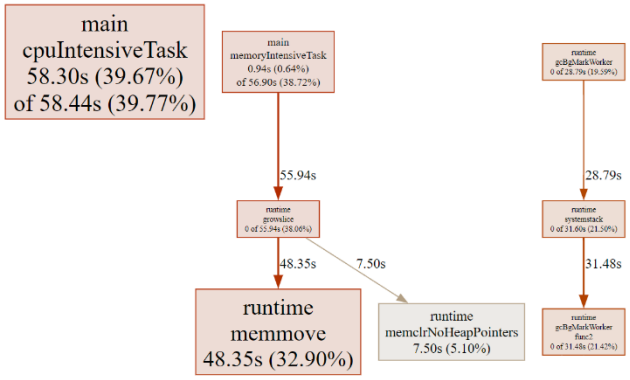

Fig. 3. 60s CPU profile genereated by pprof (truncated for simplicity).

D. *Web Interface*

In addition to the call graph generated by the command in Fig. 2, it is also possible to request a web interface to analyze generated profiles. There are multiple views allowing users to look at profile data in different ways.

E. *Other Resources*

Similar to the CPU profile shown in Fig. 3, several other resources can be monitored. This includes heap, allocs, mutexes and blocks.

V. CAVEATS OF PROFILING TOOLS

A. *Overhead*

Profiling tools can add some performance overhead due to work required for generating profile data. In some extreme cases, this may also affect the performance and resource utilization being observed.

B. *Accuracy*

Profilers usually present approximations and not exact results. The exact behavior of the system may vary somewhat from the data presented by profiling tools.

C. *Environment*

Profilers may present different data in different environments, because the application itself may behave differently. It is important to do profiling in the same or similar environment and load where the application is going to be deployed.

D. *Complex Interpretations*

In a real-world application, call graphs are get quite large, making them difficult to read. In addition, it is important to learn how to correlate observed data to root causes. Sometimes these root causes are not obvious from profile data.

CONCLUSION

Production behavior of real-world applications can be hard to predict. Profiling tools bring visibility into the system behavior which is unlikely to be obtained from other debugging and monitoring tools. They tell us how different parts of code utilize different resources. Pprof is really easy tool to use for Golang application. It provides information about a lot of different resources during runtime. However, as with any other tool, profilers are not suited to satisfy every debugging and monitoring requirement. Knowing about their strengths and limitations allow effective use of profiling tools to get deeper insights about an application's behavior.

REFERENCES

[1] "pprof Documentation (Dec 2021)," https://pkg.go.dev/net/http/pprof

[2] "pprof (Dec 2021)," https://github.com/google/pprof

[3] Prathitha Iyengar, "All You Need to Know About Code Profiling Tools and How to Choose One (Dec 2021)," https://www.headspin.io/blog/all-you-need-to-know-about-code-profiling-tools-and-how-to-choose-one

[4] Scott Gangemi, "Analyzing and improving memory usage in Go (Jul 2021)," https://medium.com/safetycultureengineering/analyzing-and-improving-memory-usage-in-go-46be8c3be0a8

[5] Mark Gritter, "Taming Go's Memory Usage, or How We Avoided Rewriting Our Client in Rust (Sep 2021)," https://www.akitasoftware.com/blog-

posts/taming-gos-memory-usage-or-how-we-avoided-rewriting-our-client-in-rust

[6] Alexandra, "What is Code Profiling? Learn the 3 Types of Code Profilers (May 2020)," https://stackify.com/what-is-code-profiling