
go-web-framework-benchmark

This benchmark suite aims to compare the performance of Go web frameworks. It is inspired by Go HTTP Router Benchmark but this benchmark suite is different with that. Go HTTP Router Benchmark suit aims to compare the performance of **routers** but this Benchmark suit aims to compare whole HTTP request processing.

Last Test Updated: 2020-05

test environment

- CPU: KVM Virtual CPU version(2 GHz, 4 cores)
- Memory: 16G
- Go: go1.18.5 linux/amd64
- OS: Ubuntu 22.04.1 LTS with Kernel 5.15.0-41-generic

Tested web frameworks (in alphabetical order)

Only test those webframeworks which are stable

- atreugo
- baa
- beego
- bone
- chi
- clevergo
- default http
- denco
- don
- echo
- fasthttp-routing
- fasthttp/router
- fasthttp
- fiber
- gear
- gearbox
- gem
- gin
- goframe
- go-ozzo

-
- go-restful
 - go-tigertonic
 - goji
 - goji
 - golf
 - gorilla
 - gorouter
 - goyave
 - httprouter
 - httptreemux
 - indigo
 - lars
 - lion
 - macaron
 - muxie
 - negroni
 - pat
 - pulse
 - pure
 - r2router
 - tango
 - tinyrouter
 - treemux
 - violetear
 - vulcan
 - webgo

some libs have not been maintained and the test code has removed them

Motivation

When I investigated performance of Go web frameworks, I found Go HTTP Router Benchmark, created by Julien Schmidt. He also developed a high performance http router: httprouter. I had thought I got the performance result until I created a piece of codes to mock the real business logics:

```
1 api.Get("/rest/hello", func(c *XXXXX.Context) {
2     sleepTime := strconv.Atoi(os.Args[1]) //10ms
3     if sleepTime > 0 {
4         time.Sleep(time.Duration(sleepTime) * time.Millisecond)
5     }
```

```
6
7     c.Text("Hello world")
8 }
```

When I use the above codes to test those web frameworks, the token time of route selection is not so important in the whole http request processing, although performance of route selection of web frameworks are very different.

So I create this project to compare performance of web frameworks including connection, route selection, handler processing. It mocks business logics and can set a special processing time.

The you can get some interesting results if you use it to test.

Implementation

When you test a web framework, this test suit will starts a simple http server implemented by this web framework. It is a real http server and only contains GET url: “/hello”.

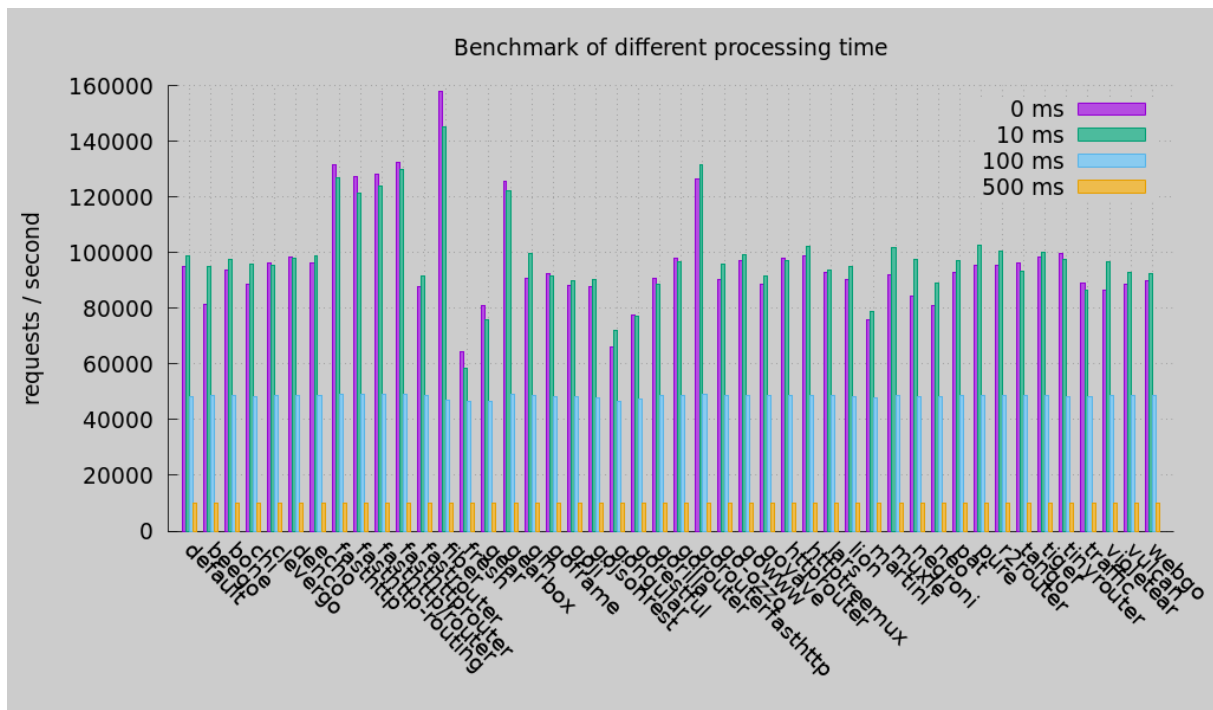
When this server processes this url, it will sleep n milliseconds in this handler. It mocks the business logics such as: * read data from sockets * write data to disk * access databases * access cache servers * invoke other microservices *

It contains a test.sh that can do those tests automatically.

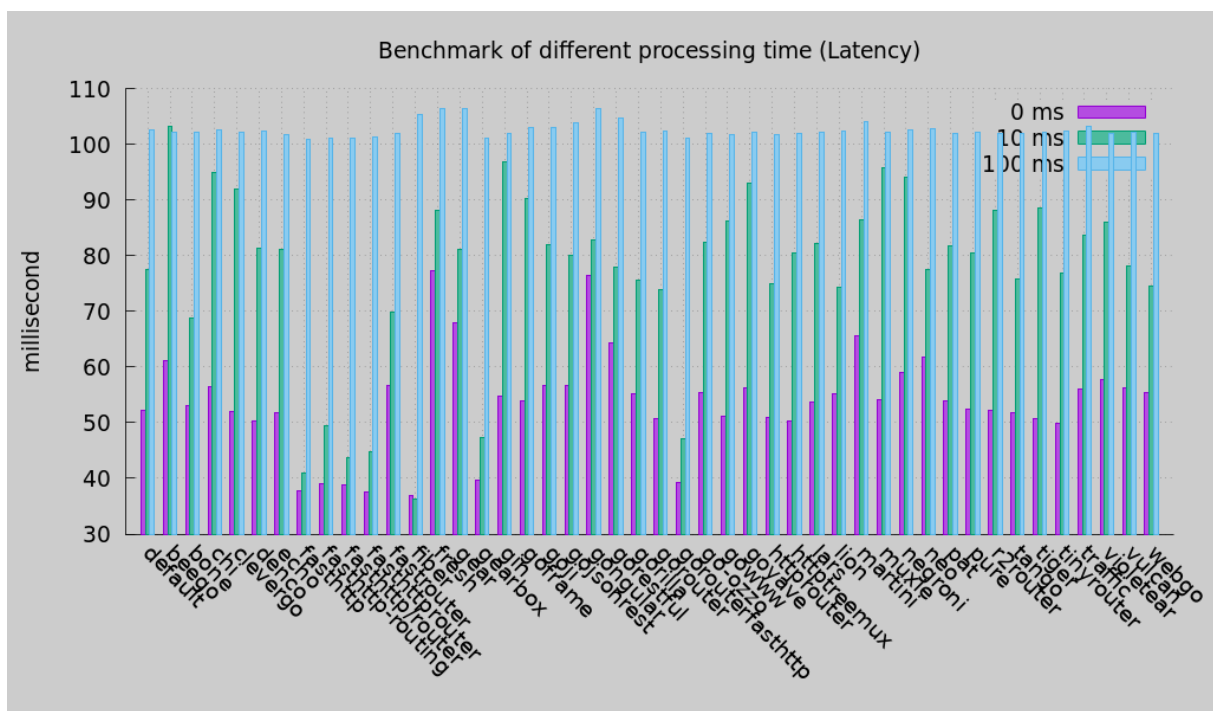
It uses wrk to test.

Basic Test

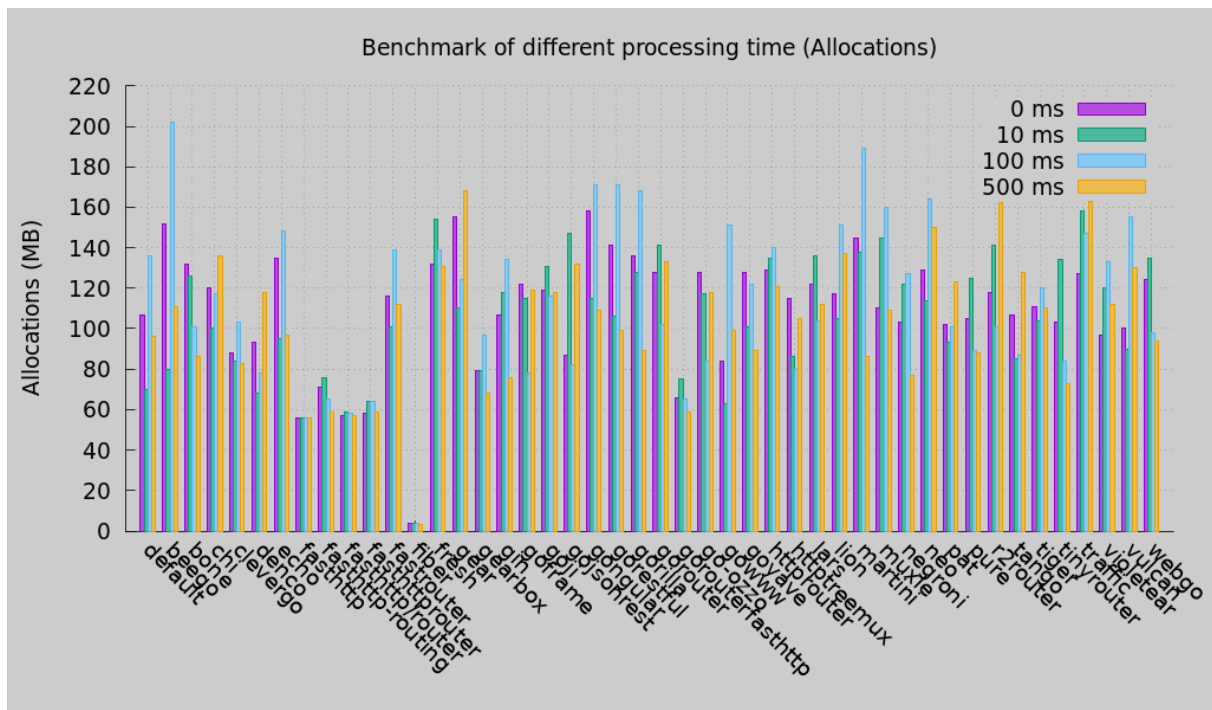
The first test case is to mock 0 ms, 10 ms, 100 ms, 500 ms processing time in handlers.



the concurrency clients are 5000.

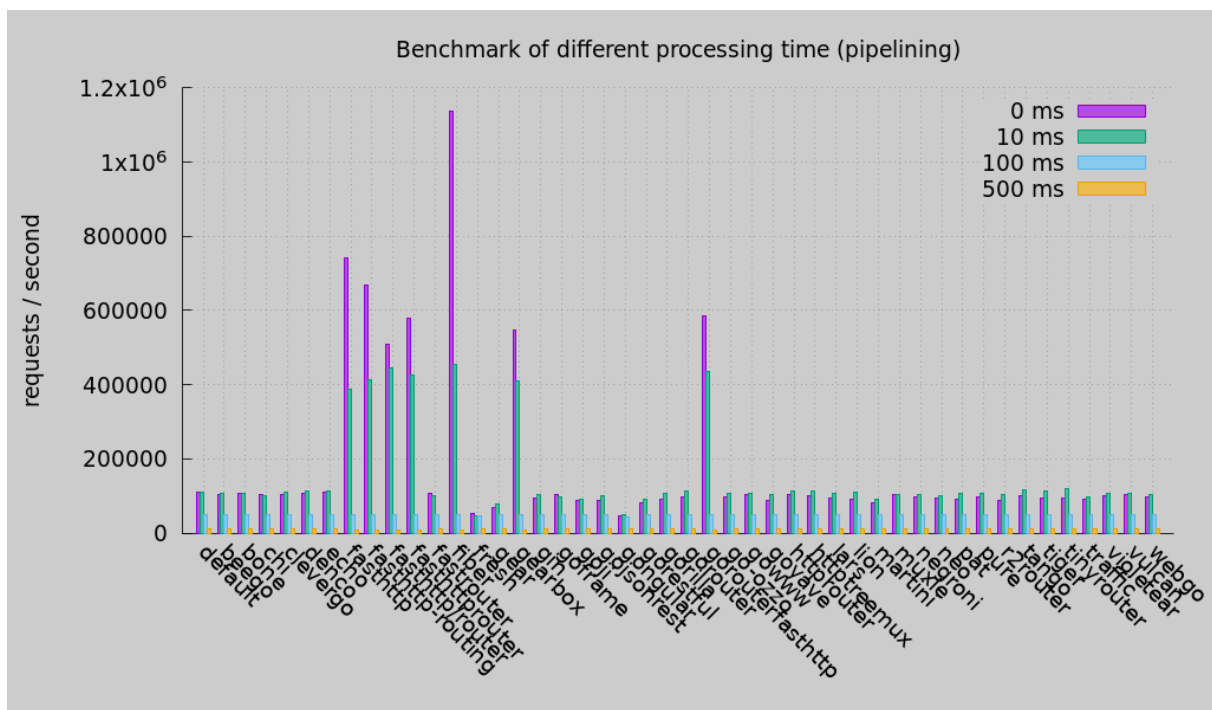


Latency is the time of real processing time by web servers. The smaller is the better.



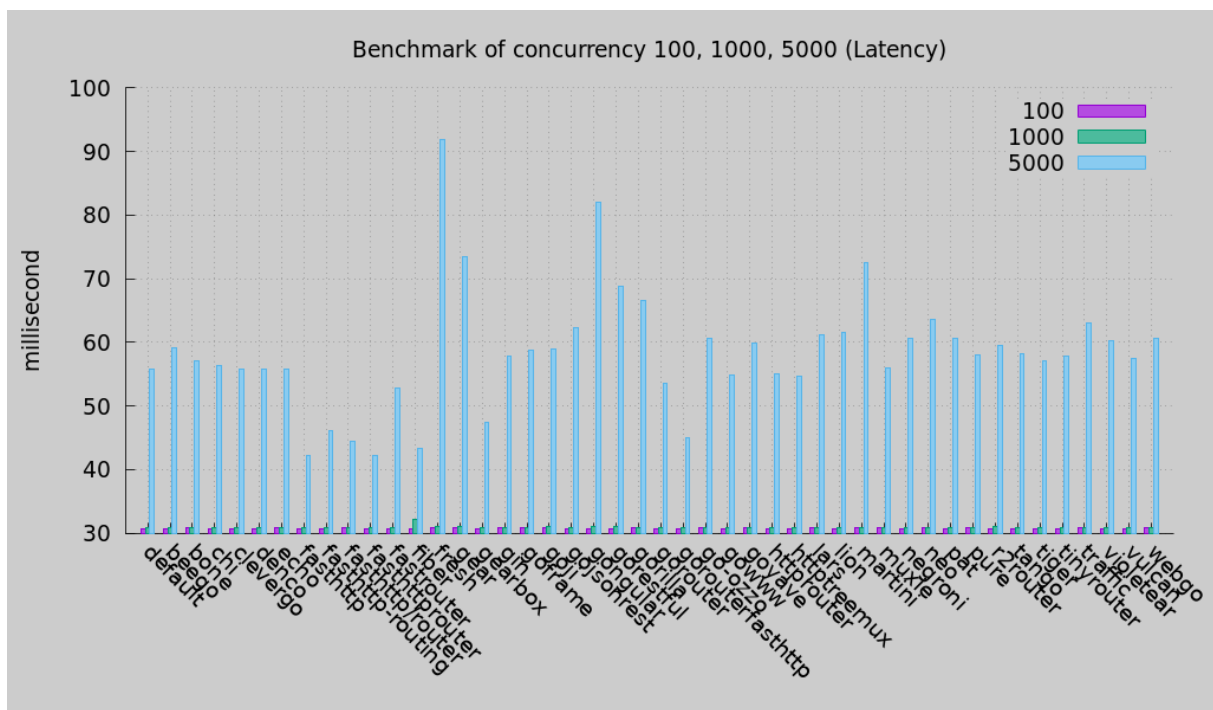
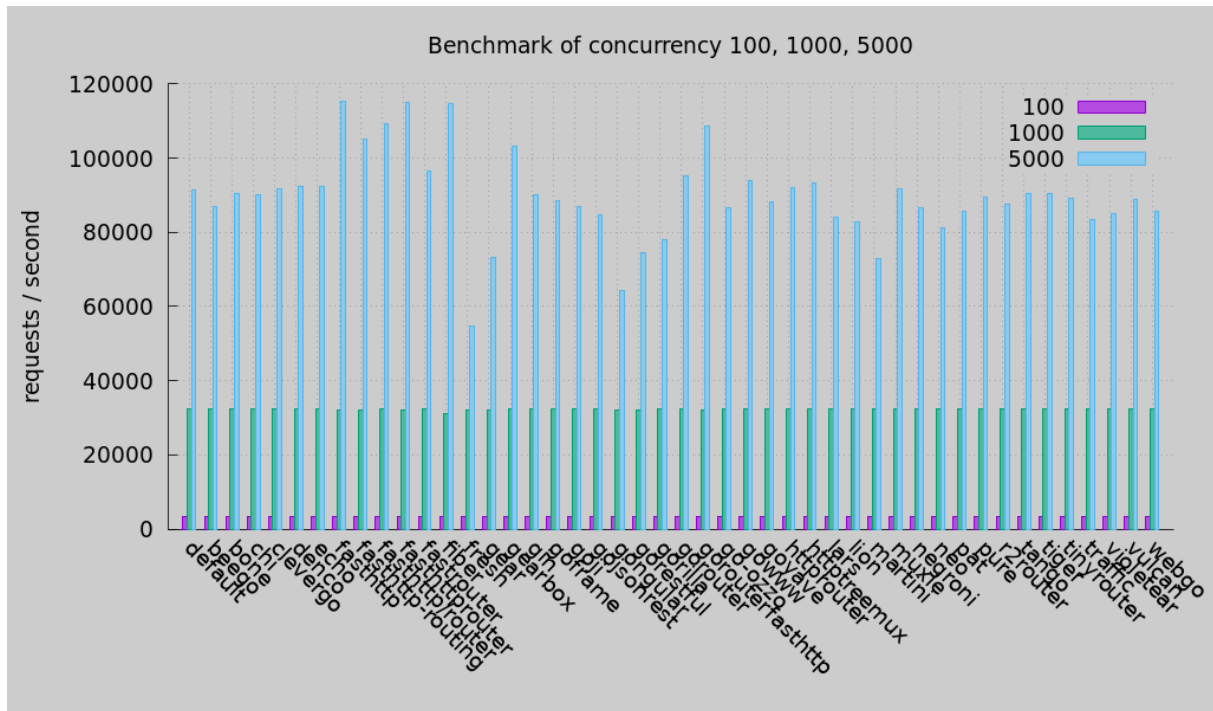
Allocs is the heap allocations by web servers when test is running. The unit is MB. The smaller is the better.

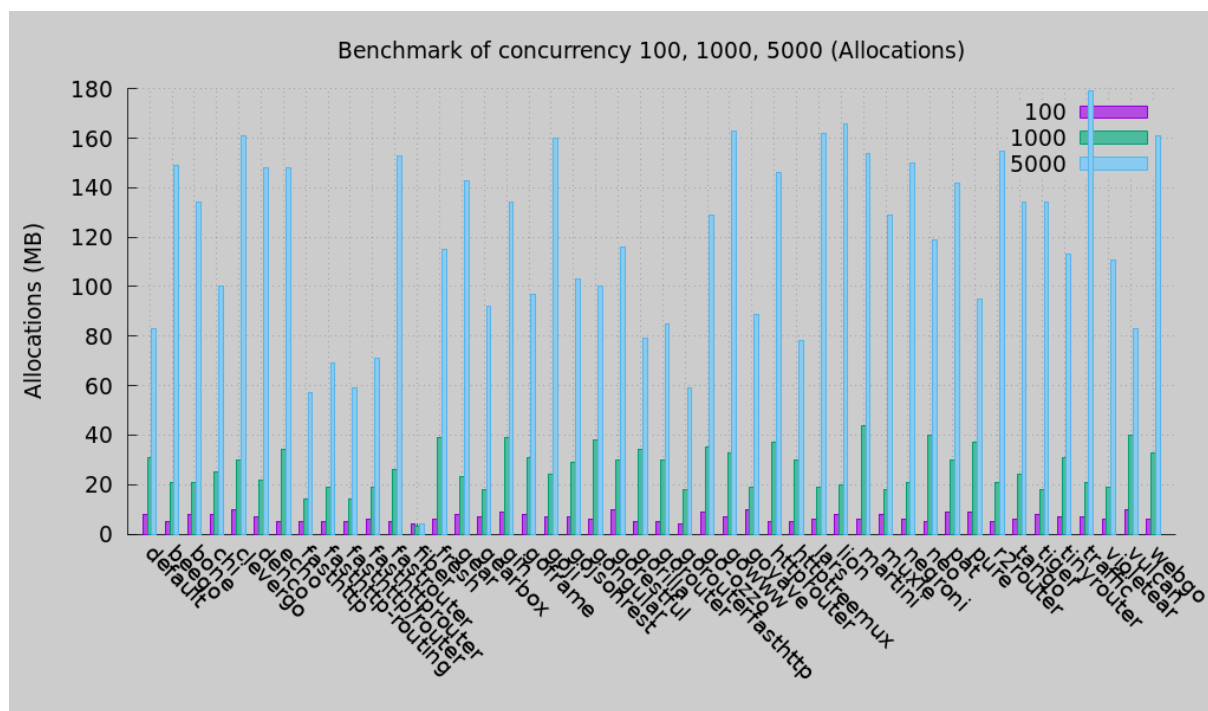
If we enable http pipelining, test result as below:



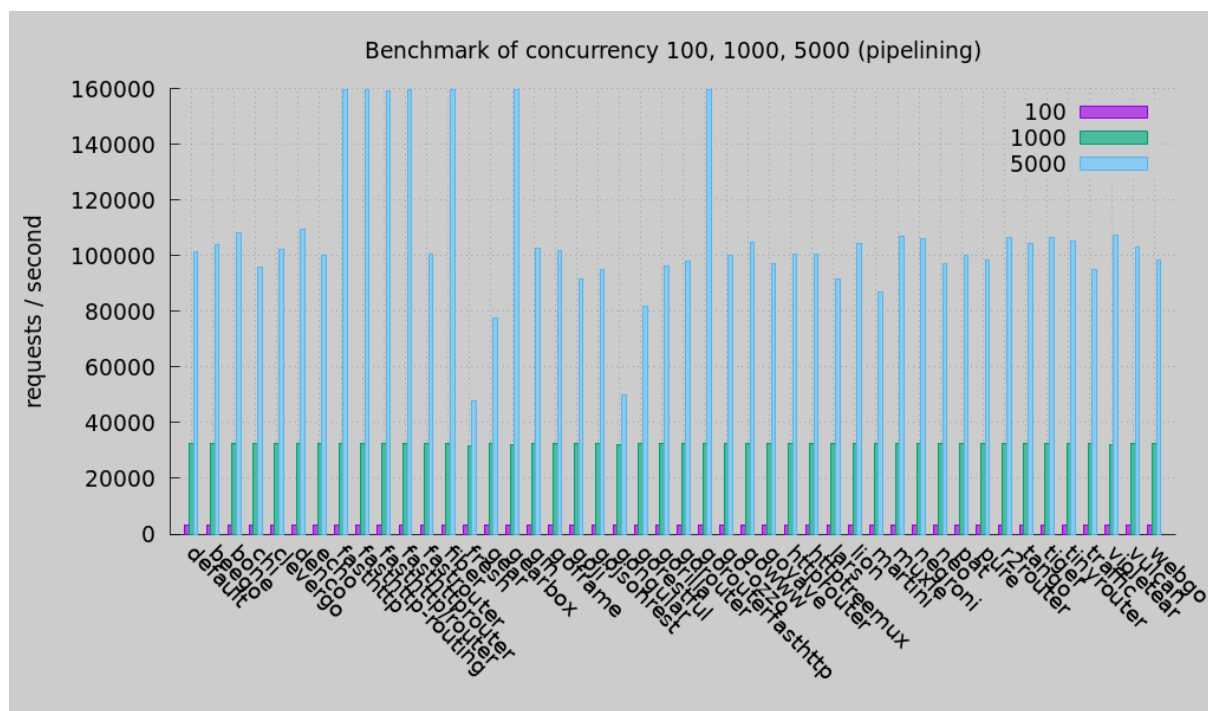
Concurrency Test

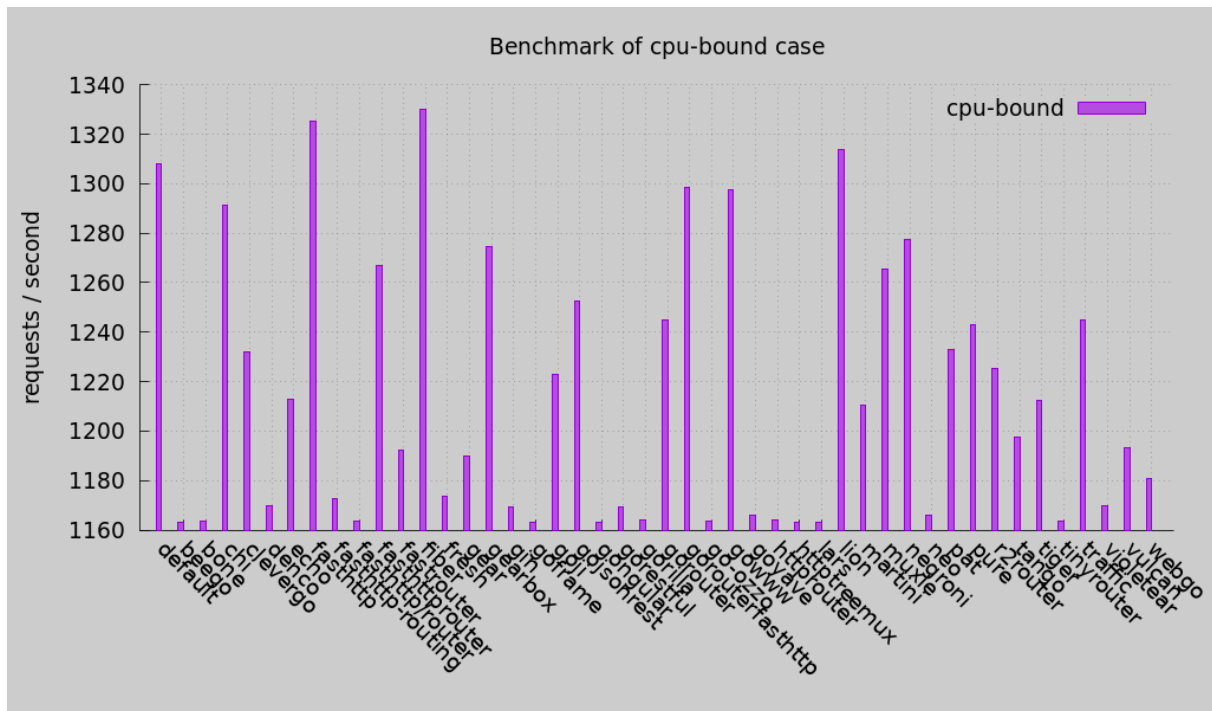
In 30 ms processing time, the test result for 100, 1000, 5000 clients is:





If we enable http pipelining, test result as below:





Usage

You should install this package first if you want to run this test.

```
1 go get github.com/smallnest/go-web-framework-benchmark
```

It takes a while to install a large number of dependencies that need to be downloaded. Once that command completes, you can run:

```
1 cd $GOPATH/src/github.com/smallnest/go-web-framework-benchmark
2 go build -o gowebbenchmark .
3 ./test.sh
```

It will generate test results in `processtime.csv` and `concurrency.csv`. You can modify `test.sh` to execute your customized test cases.

- If you also want to generate latency data and allocation data, you can run the script:

```
1 ./test-latency.sh
```

- If you don't want use keepalive, you can run:

```
1 ./test-latency-nonkeepalive.sh
```

- If you want to test http pipelining, you can run:

```
1 ./test-pipelining.sh
```

- If you want to test some of web frameworks, you can modify the test script and only keep your selected web frameworks:

```
1 .....
2
3 web_frameworks=("default" "atreugo" "beego" "bone" "chi" "denco" "don"
  "echo" "fasthttp" "fasthttp-routing" "fasthttp/router" "fiber" "gear"
  " "gearbox" "gin" "goframe" "goji" "gorestful" "gorilla" "gorouter"
  "gorouterfasthttp" "go-ozzo" "goyave" "httprouter" "httptreemux" "
  indigo" "lars" "lion" "muxie" "negrioni" "pat" "pulse" "pure" "
  r2router" "tango" "tiger" "tinyrouter" "violetear" "vulcan" "webgo")
4 .....
```

- If you want to test all cases, you can run:

```
1 ./test-all.sh
```

NOTE: comparing 2 webframeworks consumes approx. 11-13 minutes (doesn't depend on the machine). Just `test.sh` with all the webframeworks enabled will take a couple of hours to run.

Plot

All the graphs are generated automatically as the `./test.sh` finishes. However, if the run was interrupted, you can generate them manually of partial data by executing `plot.sh` in testresults directory.

Add new web framework

Welcome to add new Go web frameworks. You can follow the below steps and send me a pull request.

1. add your web framework link in README
2. add a hello implementation in server.go
3. add your webframework in libs.sh

Please add your web framework alphabetically.