Team A
Task 1

Dan Bachar Yichao Zhu Georgiy Nefedov

1 Basic Task

1.1 Description

The essence of the first task is to master creating reproducible experiments using containers. In particular, this includes setting up several containers to host a WordPress blog, importing dummy data, setting up a Rest authentication plugin, running multiple HTTP benchmarks, and finding bottlenecks with full-system profiling.

1.2 Setup

We decided to use the Docker container engine to make make our results reproducible for others. In the `docker-compose.yml` file we defined a MySQL database from the `mysql:5.7` image, a WordPress blog from the `wordpress:php7.3-fpm-alpine` image and a nginx web application from the `nginx:alpine` image. For API authentication we installed the miniOrange API Authentication plugin and configured Basic Auth.

The light-weighted alpine-based images of `nginx` and `wordpress` only occupy 22.9MB and 264MB respectively. In order to persist the data and configuration files, we specify volumes on each container. The container-internal `nginx` port is mapped to the host port 8080, and it serves as a reverse proxy to the `wordpress` container whose port is 9000. The web server `php-fpm` will get requests from `nginx`, spawn working sub-processes, and interpret the Wordpress php scripts. The `mysql` database is working at port 3306.

2 Benchmarking

We benchmarked the request throughput of our WordPress blog setup by performing `GET` requests for retrieving the home page, and `POST` requests for adding a new page to the blog. To do this we tried out many different http benchmarking utilities. httperf turned out to be the most suitable. We also ran our tests with another tool, `wrk`, and the results were similar. The

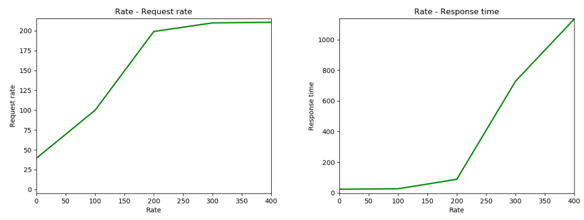

Figure 1: default state benchmark

exact command to reproduce the benchmarks and the results can be found here.

2.1 Retrieve the home page

Using `httperf` we retrieved the front page `http://ryan.dse.in.tum.de:8080/` 1000 times. For simplicity, the rate is set to 0, it means that the connection will be generated sequentially (a new connection is initiated as soon as the previous one completes) and got the throughput `264.2 req/s` or `3.8 ms/req` with request size `71 B`.

2.2 Create a page using the Rest API

Please install and activate the `miniOrange API Authentication` plugin in Wordpress to reproduce this part. Also with `httperf` we sent 1000 POST requests to `http://ryan.dse.in.tum.de:8080/wp-json/wp/v2/pages` with the contents from this file to create a new blog page. The content size is about 6kB.
The throughput was `5.3 req/s` or `190.0 ms/req` with request size `6968 B`, about 50 times less then the simple GET request.

2.3 Profiling

We use some full-system profiling tools `perf, bcc` to analyse the bottlenecks in the system and generate the Flamegraph at figure 2. It shows that the `php-fpm` process takes

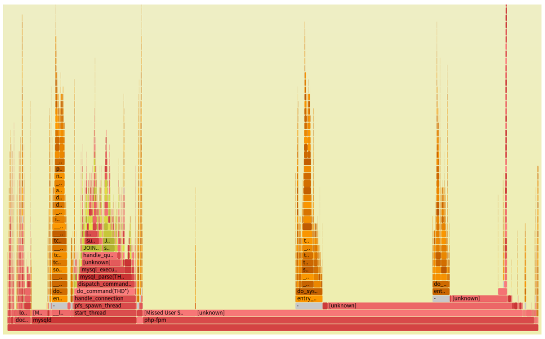
Figure 2: Flamegraph

much more CPU time than `mysqld`. There are two visible peaks, and five hot kernel space functions, from `php-fpm`. These kernel-space functions undertake the responsibility of communication. Most of the time is consumed by the user-space functions, which do jobs like scripts interpretation and process management. We thus needed to find other ways of improving our performance, instead of trying to change the source code. The details of this part can be found here.

2.4 Result

The initial version of our deployment, before the exploration task, reached 210 requests per second, with an average of 4.7 ms per request. The strong differences with the forthcoming improved version are especially felt here, with an improvement of over 16600% in throughput.

3 Exploration Task

3.1 Description

The main objective of the exploration task is to optimise throughput and to demonstrate the effect of each optimization on throughput. It involves analysing system bottlenecks with the help of system profiling tools and trying to optimise them at various levels such as application server settings, caching, resource allocation, etc.

3.2 Setup and Methodology

We first increased the load on the system by using the http load testing method in the previous section with and at the same time used the system monitor program(`top, sysstat, ss, docker stats`) to analyse the current usage of system resources such as cpu and memory, and analysed the hot functions in both user space and kernel space with the `bcc` system profiling tool. It was found that each `php-fpm` working process had enough CPU time to work, but the overall system performance was not fully utilized, so more child processes could still be forked to handle requests. At the same time, concurrent repeated requests would result in too many unnecessary database accesses, so the cache mechanism is used to increase the system.

3.3 Result

The original version of the nginx-php-mysql server had the throughput of `210 req/s` while requesting the front page with `GET`, after increasing `php-fpm`'s maximum amount of children to 25, it has reached a throughput of `1000+ req/s`. Replacing MySQL with MariaDB has allowed us to go beyond to `1400+ req/s`. With 70 php-fpm subprocesses, we've reached `5431.01 req/s`, as shown on figure 3. After activating the WP Super Cache plugin, we've reached a cap of `35543 req/s` requests per second compared to the `5431.01 req/s` without cache. For details, please see the document here.

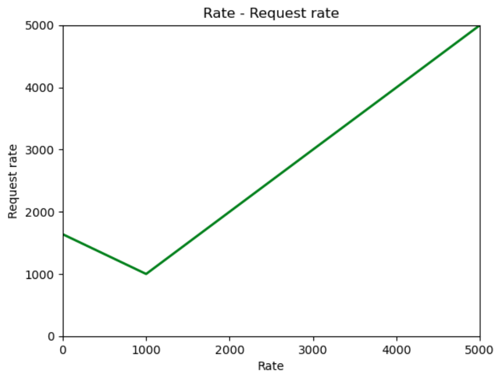

Figure 3: 70 php children with mariadb exceeds 5400 req/s

The cache plugin has enabled a great improvement in our performance benchmarks because wordpress is a great candidate for caching. All wordpress pages are dynamic and are built by the requests with appropriate parameters to a file called `index.php`. The server then builds the requested view for the user, and sends it back via the HTTP response. Since all of the benchmark requests request the same page, the plugin just has to create this file once, and statically serves this file to them, instead of going through the relatively expensive process of dynamically creating the requested view every time.
The database replacement came to mind because it is extremely easy, as MariaDB is a drop-in replacement for MySQL.
Changing `php-fpm`'s `pm` setting to static had also a noticeable effect since instead of waiting for the child processes to be dynamically created as demand arrives, the child processes are statically available initially.

References

[1] https://docs.docker.com

[2] https://www.brendangregg.com/flamegraphs.html

[3] https://linux.die.net/man/1/httperf

[4] https://medium.com/swlh/wordpress-deployment-with-nginx-php-fpm-and-mariadb-using-docker-compose-55f59e5c1a

[5] https://github.com/danbachar/swiss-knife/tree/master/task1