COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

PERFORMANCE AND SPEED OPTIMIZATIONS OF WORDPRESS-BASED WEB APPLICATION AND ITS UNDERLYING SERVER STACK

Bachelor's thesis

2015 **Rastislav Lamoš**

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

PERFORMANCE AND SPEED OPTIMIZATIONS OF WORDPRESS-BASED WEB APPLICATION AND ITS UNDERLYING SERVER STACK

Bachelor's thesis

Study program: Applied computer science
Supervisor: Mgr. Kamil Maráz.

Bratislava 2015 **Rastislav Lamoš**

Declaration of authorship

I hereby declare and confirm that this thesis is entirely the result of my own work except where otherwise indicated.

Bratislava, 31. May 2015 .....

Acknowledgement

To be written...

Abstract

In our work, we will be dealing with the performance and speed optimisations of an underlying server and a WordPress-based web application running on it. After a thoughtful study of our work, an ordinary web programmer or administrator will be able to install and configure his web server and refactor the source code of his PHP (WordPress) web application, without having to research what to modify or avoid from other sources, thus saving his time and resources. Product of our work will be a highly optimised server with proper caching and a WordPress application developed to be as efficient as possible.

Keywords: *page loading speed, WordPress, optimisation, PHP, server*

Abstrakt

V našej práci sa budeme zaoberať optimalizáciou výkonu a rýchlosti serveru a web aplikácií založenej na systéme WordPress, ktorá na tomto serveri beží. Po pozornom prečítaní a naštudovaní našej práce si bude bežný web programátor či administrátor vedieť nainštalovať a nakonfigurovať svoj web server a refaktorovať zdrojový kód svojej PHP (WordPress) web aplikácie bez toho, aby musel z iných zdrojov zisťovať čo zmeniť a čomu sa vyvarovať, čím sa ušetrí jeho čas a zdroje. Výsledkom našej práce bude vysoko optimalizovaný server s vhodným caching a WordPress aplikácia vyvinutá tak, aby bola čo najefektívnejšia.

Kľúčové slová: *rýchlosť načítavania stránky, WordPress, optimalizácie, PHP, server*

Glossary

caching test 13

Contents

Glossary **7**

1 Introduction **11**
1.1 Problem Definition 11
1.2 About WordPress 12
1.2.1 General Information 12
1.2.2 WordPress-Powered Website 12
1.3 General Techniques for Improving Website Loading Speed 13
1.3.1 Web-Serving Software Efficiency 13
1.3.2 Server-Side Caching 13
1.3.3 Client-Side Caching 15
1.3.4 JavaScript and CSS Resource Minification and Combination 15
1.3.5 Compression of Images, HTML and Other Resources 16
1.3.6 Optimizing Images — Image Sprites 16
1.3.7 CDN and Resource Distribution 17
1.4 Similar Studies 17
1.4.1 Using Nginx, Apache, APC and Varnish in Different Scenarios — Garron.me 17
1.4.2 WordPress on HHVM vs WordPress on PHP-FPM — WPengine.com 18
1.4.3 WordPress HHVM vs PHP — xyu.io 19

2 Configuring testing environment **20**
2.1 Server parameters 20
2.2 Ansible automation 20
2.3 Installing required software on the server 21
2.4 Using loader.io for load testing 22

3 Benchmarking server software **23**
3.1 Apache with mod_php 23
3.1.1 Load testing Apache with mod_php 23

3.2 Nginx with PHP-FPM 26
3.2.1 Load testing Nginx with PHP-FPM 27
3.3 Nginx + HHVM 28
3.3.1 Load testing Nginx with HHVM 29
3.3.2 Advanced WordPress and HHVM 30

4 Caching **32**
4.1 Database caching 32
4.2 Page caching 33
4.3 Browser caching 35

5 Source code performance optimizations **37**
5.1 Profiling source code with XHProf 37
5.2 Source code optimization techniques and best practices 38
5.2.1 Autoloading PHP Classes 38
5.2.2 Time complexity of PHP functions 38
5.2.3 WordPress Plugin API 38
5.2.4 WordPress database queries and structure 39

6 Concluding remarks **41**

7 Future work **42**
7.1 Load balancing 42
7.2 Perception of speed 42

References **47**

List of Figures

1.1 Slower page response time results in an increase in page abandonment . . . 11
1.2 What can a one-second page delay cause to your e-commerce site? 11
1.3 PHP execution diagram with and without APC opcode cache. 14
1.4 Apache HTTP + PHP, no opcode caching 18
1.5 Apache HTTP + PHP, APC opcode caching 18
1.6 WordPress on PHP vs WordPress on HHVM response times 19

3.1 Apache HTTP with mod_php: clients versus average response time 24
3.2 Apache HTTP with mod_php: Htop process viewer 2 seconds into test . . . 25
3.3 Screenshot of Htop's help screen explaining the main gauges 25
3.4 Apache HTTP with mod_php: Htop process viewer 25 seconds into test . . 25
3.5 Nginx with PHP-FPM: clients versus average response time 27
3.6 Nginx with PHP-FPM: Htop process viewer 1 second into test 28
3.7 Nginx with PHP-FPM: Htop process viewer 22 seconds into test 28
3.8 Nginx with HHVM: clients versus average response time 29
3.9 Nginx with HHVM: Htop process viewer 1 second into test 30
3.10 Nginx with HHVM: Htop process viewer 22 seconds into test 30
3.11 Nginx with HHVM and advanced WordPress: clients versus average response time 31

4.1 Nginx with FastCGI caching: clients versus average response time 34
4.2 Nginx with FastCGI caching: Htop process viewer 9 seconds into test . . . 35

1. Introduction

1.1 Problem Definition

Website loading speed has a critical impact on page abandonment rate among its visitors. According to a Google experiment, website loading time worsened only by **half of a second** had a 20% drop in its visitor's traffic. [10] From the figure 1.1 [53], we can see a chart of page abandonment relative to a web page loading time in seconds. By the time a ten-second-long web page is loaded to its visitors, more than 35% of them will close the page, never seeing what is on it.

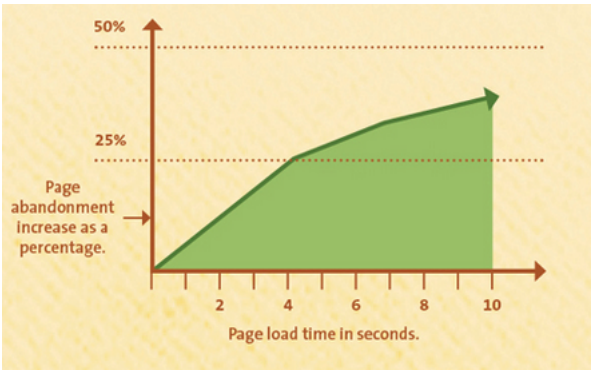

Figure 1.1: Slower page response time results in an increase in page abandonment

To highlight the seriousness of this issue, imagine a situation in which an e-commerce site owner is making $100,000 per day. One-second page delay could potentially cost him or her $2.5 million in lost sales every year [53]. See figure 1.2 for more detail.

Figure 1.2: What can a one-second page delay cause to your e-commerce site?

What is more, Google incorporated site speed in search rankings in 2010 [4], meaning that the more time it takes to load a website, the lower it ranks in Google search results. We

believe we have shown the reader enough evidence that optimizing the performance and speed of a website is crucial for its success, especially in today's fast-paced world.

1.2 About WordPress

1.2.1 General Information

Citing WordPress.org, "WordPress is web software you can use to create a beautiful website or blog."[47] Simply put, WordPress is a powerful, open-source web publishing software, content management system and a web platform for building rich web applications. People and companies use WordPress for various purposes and reasons, most notably for their blogs, websites, e-commerce solutions and large web portals. At the time of writing, it is estimated that about 24% [38] of all websites, whose content management system is known, are being run on WordPress. This number is rather astonishing because it means that visiting five random websites, one of them will be a WordPress-powered one.

WordPress, in comparison to other web software and frameworks, is a full-featured, stand-alone web publishing software and content management system. WordPress is written mostly in PHP language, accompanied with JavaScript scripts and CSS stylesheets. At its core, it consists of a request-response routing subsystem, classes for managing database and content, security features and others. Initially, WordPress was started as an open-source hobby project of Matt Mullenweg in 2003 [49]. Since then, web programmers from all around the world have contributed to it, making it robust, secure and fast. However, as there are several bottlenecks in the system, we, web developers, decided to analyze and improve them. Our findings and results are summarized in this thesis.

1.2.2 WordPress-Powered Website

In order to customize the look and feel of user's instance of the website, there exists a mechanism called the WordPress Theme. A WordPress theme is simply a collection of scripts, stylesheets and images which get combined, processed and the generated content is sent back to the client. A user can install any theme which is compatible with his or her WordPress version. There is no central body governing the quality and correctness of a theme. As theme developers are free to build them almost arbitrarily, numerous security and performance flaws occur. The WordPress Plugin is a pluggable piece of software which enhances the basic WordPress functionality, thus enabling its users to heavily modify their WordPress-based web applications and sites. Plugins are also open source and without any

quality guarantees, therefore experiencing the same problems as the aforementioned themes.

Moreover, when a user has a high-traffic website, his web delivering server is not able to keep up with all the requests resulting in a slow, unresponsive website. The reader might assume that the problems would be solved by increasing plugins quality. While the previous statement is true, it is usually not viable, mainly due to a reason that work of an experienced web developer is relatively expensive. In many cases, upgrading the server and/or getting additional one(s) is the preferred way. In our work, we are concerned with optimizing the server software first and only then examining the best practices, tips and tricks of a plugin or a theme development.

1.3 General Techniques for Improving Website Loading Speed

Before we can delve into solutions to our problems, we need to list and describe several general techniques and measures used to improve not only the website loading times, but also a server resources usage and load.

1.3.1 Web-Serving Software Efficiency

Using the most performant and the least resource-hungry web-serving software is usually the most effective way of improving web page loading speed. Some studies have found that even small changes to configuration files can a difference. In our work, we are making comparisons between the two most popular web-serving applications, namely Apache HTTP and Nginx, as well as comparisons between different PHP interpreters. Jump to section **??** for more details and statistics.

1.3.2 Server-Side Caching

Let us divide server-side caching into three main categories:

1. **Caching intermediate PHP code**
2. **Caching the output of PHP interpreter (so called "page cache")**
3. **Static resources (files) caching**
4. **Database caching**

Caching intermediate PHP code

PHP source code is interpreted on each run, thus the PHP interpreter has to read and collect all required files each time a new request is sent to our WordPress-based website. With PHP opcode caching, an intermediate source code is generated on the first run and stored temporarily. When a new request comes, instead of going through all the files, PHP loads cached opcode and executes it. We can observe the process from the figure 1.3 [28].

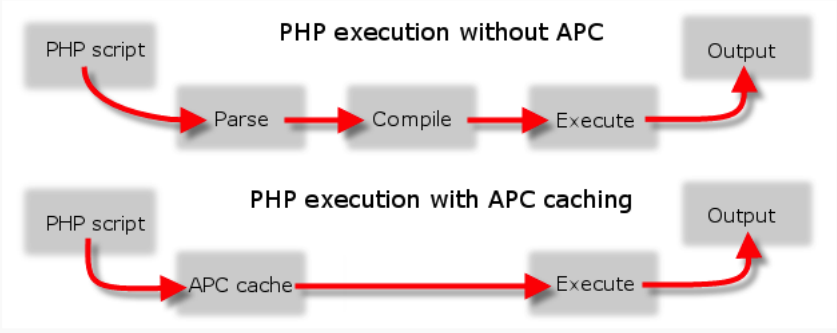

Figure 1.3: PHP execution diagram with and without APC opcode cache.

Caching the output of PHP interpreter (so called "page cache")

While a website based on WordPress is dynamic, in some cases it is possible to store the constructed web page for subsequent usage. This process is usually called page caching and it is suitable for websites with static blocks of content. However, problems arise when we have to handle special cases such as when a visitor is logged into our website. In that case, we are not able to cache the request because other visitors would see the logged-in visitor's cached content instead of a general one.

Static resources (files) caching

When our website consists of a large number of (smaller) files such as JavaScript scripts, CSS stylesheets, images and others, caching these resources is an idea worth mentioning. They are constructed and loaded on the initial request, stored in a local web server cache and retrieved from the cache on subsequent requests.

The largest disadvantage of static file caching is usually a drop in free memory available to different needs of the web server, especially when files are cached in RAM memory.

Database caching

When multiple requests to our web server trigger the same database queries, it is reasonable to store the retrieved data for later use. When a subsequent request is made, querying the

database is omitted, thus preserving valuable server resources and outputting the resulting web page faster.

1.3.3 Client-Side Caching

What makes client-side caching different from the server-side caching is that data is stored locally in the visitor's browser, not on the server to which the requests are made. [30] The whole hypertext document with all its resources including JavaScript scripts and CSS stylesheets can get cached in client's browser storage, loaded from it on consecutive requests. Naturally, the most notable advantage of client-side caching is the fact that the user's browser does not need to download the locally cached resources.

On the other hand, we need to implement a mechanism which can handle resource changes on the server-side. When not done properly, some of the visitors might see outdated content due to the reason that the visitor's browser has not been instructed to revalidate its cache.

1.3.4 JavaScript and CSS Resource Minification and Combination

At the time of writing (early 2015), a large number of WordPress themes and plugins contain tens or more JavaScript scripts and CSS styles, especially the more professional ones. It is caused by the fact that users like having amazingly looking, feature-rich websites. The problem with this fact is twofold:

- **There is a limit on the number of concurrent downloads of resources from the same domain. Both Internet Explorer 8 and Google Chrome allow six concurrent downloads while Firefox eight. [54]**
- **Each request carries an overhead of constructing a HTTP packet, sending it to the web server and waiting for the reply.**

There are two well-known methods of solving this issue:

1. **Resources combination and/or**
2. **resources minification.**

Resources combination

Resource combination is a process in which resources on the requested web page are collected into (preferably) single file which is then sent back to the visitor's browser. There exist two main downsides of this approach. The first is that collecting the resources consumes

additional CPU cycles on the server-side. Another disadvantage happens when the owner of the website modifies source code of any of the grouped resource. The whole group has to be gathered together again, including revalidating browser cache if present.

Resources minification

JavaScript scripts and CSS stylesheets can be minified. Minification is a procedure in which parts of a script or a stylesheet are reduced or completely removed, thus reducing its size and length. Advanced minification tools are capable of refactoring them in a manner that they become even more compact.

1.3.5 Compression of Images, HTML and Other Resources

Image compression

Images, particularly those produced by the JPEG lossy compression mechanism, can be compressed further, thus reducing their size while keeping a tolerable quality of picture details. At the time of writing this work, the cost of satisfying WordPress plugin [29] for image compression is zero.

Hypertext documents and other text-based assets compression

Before the data is sent back in a response to visitor's request, its size can be reduced further with a process called data compression. [8] Most modern web browsers [32] support GZIP compression of textual data. One of the downsides of performing this process is that additional CPU cycles on both server and client sides are expended in order to compress and uncompress the data.

1.3.6 Optimizing Images — Image Sprites

Images used for our website's user interface as well as other images can become more optimized for use in the web environment. A mechanism called image spriting [35] is a procedure during which multiple images, sometimes even all of them, get collected and combined into a single larger image — sprite. When a web page is requested, instead of loading tens of user interface icons and images, only this single one is sent back to the user, thus saving additional HTTP requests and bandwidth. When rendering the user interface, icons and images are taken from that single image.

1.3.7 CDN and Resource Distribution

In some cases, our website is accessed from many different countries, even continents. If we have web servers located only locally, additional milliseconds start to congregate in these situations. To solve this problem, web administrators usually distribute the website's data across multiple servers positioned in multiple places around the world. CDN services greatly reduce the amount of work needed to accomplish the goal by doing it automatically for us. Another quite useful concept is called load balancing. It is a method of distributing requests to a website across multiple web servers decreasing the overall load on each machine.

1.4 Similar Studies

In this section, we will be analyzing and discussing similar studies done by other web developers and programmers. Before we look at their results, we need to have a basic understanding of the web serving software they were comparing.

Apache HTTP, also called Apache, is a free and open source world's most widely used (57.5%[37] of all websites whose web server we know) web server software. **Nginx**, released nearly 10 years later than Apache, was designed with a high concurrency, high performance and low memory usage in mind, specifically to solve the C10K problem [11].

Both **PHP** and **HHVM** are PHP script interpreters. HipHop Virtual Machine (HHVM), released by Facebook in 2011, was developed to increase the performance of PHP script execution on the Facebook site. Both of them are open source and free to use.

1.4.1 Using Nginx, Apache, APC and Varnish in Different Scenarios — Garron.me

Guillermo Garron, the author of the work [7], has performed several comparisons between the most popular web-serving and caching software. He used a relatively weak web server with highly limited computing power — 512MB RAM, a shared CPU and a shared disk. However, the result of his experiment is clear — caching the output of the PHP interpreter has a considerable impact on loading times of WordPress-powered web page.

From the figure 1.4, we observe that if the number of simultaneous visitors exceeds ten, response times of the web server start to dramatically decrease in a linear fashion. On the other hand, response times start to worsen only after thirty concurrent visitors, as we can see from the figure 1.5.

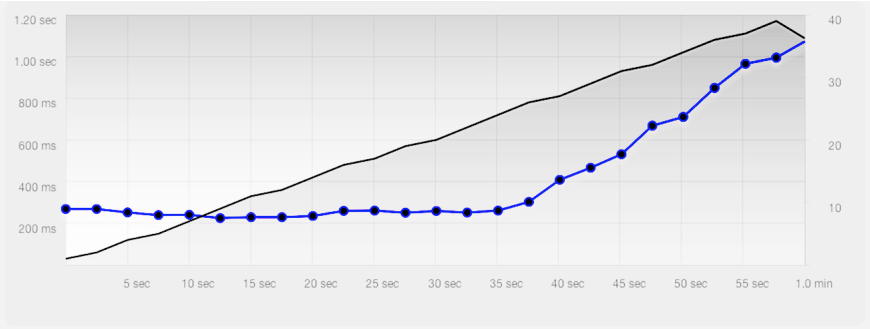

Figure 1.4: Apache HTTP + PHP, no opcode caching

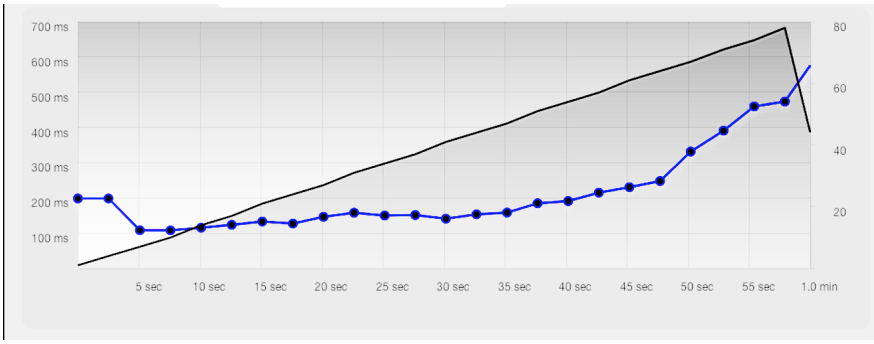

Figure 1.5: Apache HTTP + PHP, APC opcode caching

Although using Nginx instead of Apache HTTP yields additional performance gains, they are less notable. Nginx strengths are demonstrated when a website is composed of many resources. However, Garron used a standard WordPress installation, with no extra plugins or complex custom themes.

1.4.2 WordPress on HHVM vs WordPress on PHP-FPM — WPengine.com

WPengine.com is a company specializing in offering WordPress web hosting services. They have introduced a new hosting plan which differentiates itself from others by using HHVM PHP interpreter instead of the standard PHP. Before releasing the hosting plan, a study, in which the performance of HHVM versus PHP was compared, was undertaken. The study concluded with a fact that HHVM increased the speed of their servers by 600% [34]. This fact is displayed in the figure 1.6.

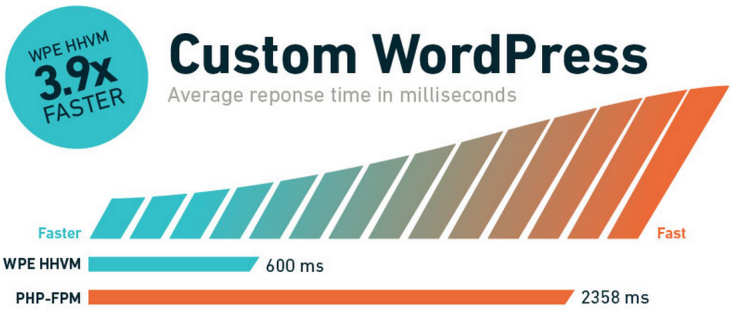

Figure 1.6: WordPress on PHP vs WordPress on HHVM response times

In the study, they found out that HHVM is not 100% stable yet and occasionally stops working. Their solution to this problem was to redirect the incoming requests to a fallback PHP interpreter while HHVM gets restarted.

1.4.3 WordPress HHVM vs PHP — xyu.io

Xiao Yu, a web developer, benchmarked [55] WordPress running on PHP vs WordPress running on HHVM. His findings show a similar pattern to the findings of WPengine.com in section 1.4.2. Outcome of his experimentation can be observed in table 1.1.

	Response Time	*Ok Responses*	*Errors / Timeouts*
Anon PHP	4.091	8,939	0.94%
Anon HHVM	2.122	18,308	0.00%
Change	48.1%	**2.05X**	
Auth PHP	20.688	457	74.17%
Auth HHVM	14.359	1,242	43.45%
Change	30.6%	**2.72X**	

Table 1.1: WordPress on HHVM vs WordPress on PHP — xyu.io

”In the numbers above anonymous requests represents hits to various pages without a WordPress logged in cookie which are eligible for Batcache caching whereas authorized requests are hits to the same pages with a login cookie thus bypassing page caching.”[55]

2. Configuring testing environment

2.1 Server parameters

The benchmarking and WordPress optimizations will be done on a Ubuntu-based VPS server. Ubuntu comes with a prebuilt software packages (through apt-get package handling utility), therefore it is quick and relatively easy to install and set up applications we will use.
We are going to use vpsfree.cz [36] as the server provider. However, content of this work can be reproduced on any kind of modern Ubuntu hosting platforms such as digitalocean.com [5].
Specification of the testing server are:

- 3x CPU
- 4GB RAM
- 300Mbps connectivity

We will be using Ubuntu version 14.04.

2.2 Ansible automation

Before benchmarking the performance of various web serving software, we have to install and configure the server. However, if done manually, it is a tedious task and takes a lot of time. One of the solutions to this problem is to use an automation tool.

Ansible is a modern IT automation tool. It is a command line application which runs from a host computer, executing commands on remote servers. The main parts of Ansible are tasks and modules.

Ansible Task has a name describing, in short, what it does. The task runs single Ansible module which performs some operations on the server. These modules are subprograms that can usually take several arguments, altering its behavior.

For better organization, tasks can be grouped into roles and playbooks. Ansible Role is a logical container for multiple related tasks aiming to accomplish a single goal, like installing and setting up Nginx. An Ansible Playbook is a file specifying roles which should be run on specific servers. These servers are listed in a hosts file located within the Ansible project directory.

Ansible files (except for the hosts) are written in the YAML syntax. It is straightforward to read and understand. We have prepared several playbooks which will provision the whole server so it can be benchmarked. They are located in the appendix of the work.

2.3 Installing required software on the server

We are going to install all the required software on our testing server. Copy the attached wordpress-ansible folder or clone its GitHub [13] repository to your local computer. In the folder, we can find a few playbooks, hosts file, variables and roles. We will be referring to this folder by the name ”wordpress-ansible” in the work.

The first step is editing the hosts file. Put the IP address or hostname of your testing server in the file and assign it to the webservers group.

Secondly, we need to install Ansible on our local computer. Official Ansible documentation [2] explains how to do it.

The next step is to edit the group variables for our testing server. Open the group_vars/webservers file and modify it to your needs. If you are satisfied with the default values, there is no need to change it.

After completing all the previous steps, navigate to the wordpress-ansible folder in your command line and execute following command:

```
ansible-playbook -i hosts install-all-software.yml
```

During the installation process, Ansible will inform you about the status of executed tasks, whether they went successfully or not.

We are now ready to benchmark web serving software.

2.4 Using loader.io for load testing

Apart from having a server running a WordPress-based site, we need a tool for load testing the server. Initially, we used a popular command line utility called ”ab” (standing for Apache Bench). However, this approach also required a second server from which the ab utility would be run. What is more, storing the tests results and plotting them on charts with ab is not trivial.

We found a simple cloud-based load testing service called ”loader.io” [22]. In a subscription, free of charge, we get 10,000 clients (requests) per test and one target host. For our purposes, these parameters are enough.

After creating an account at loader.io and adding a target host (our testing VPS server), we are presented with a dialog to verify the ownership of the server. While putting the token manually in a text file to our WordPress site directory will allow loader.io to verify the server, it is better to come up with an Ansible playbook which would automate this task for us. Below is the YAML definition file of the playbook.

```
---
- name: Loader.io host verification >

  hosts: webservers
  remote_user: root
  vars:
    token: <YOUR-TOKEN-HERE>

  tasks:
  - name: Install loader.io verification token
    shell: echo {{ token }} > {{ token }}.txt
    args:
      chdir: "{{ remote_wordpress_dir }}"
      creates: "{{ remote_wordpress_dir}}/{{ token }}.txt"
```

Replace ¡YOUR-TOKEN-HERE¿ placeholder with your verification token and save the playbook as ”loader.io-verification.yml”. Whenever we provision a new testing WordPress site, we only need to run the loader.io-verification.yml playbook to have the token inserted to the site.

3. Benchmarking server software

The first step is to compare the performance of Apache and Nginx web serving applications. However, it is not necessary to compare these applications in their speed of serving static content (for example HTML files) because it has been proven and because of Nginx architecture that Nginx is more performant in this manner. What is interesting to compare, though, is running of PHP under Apache and Nginx.

3.1 Apache with mod_php

”Apache supports a variety of features, many implemented as compiled modules which extend the core functionality.” ”Instead of implementing a single architecture, Apache provides a variety of MultiProcessing Modules (MPMs), which allow Apache to run in a process-based, hybrid (process and thread) or event-hybrid mode, to better match the demands of each particular infrastructure.” [39]

In a default configuration, when HTTP requests are received, Apache starts to process them one by one. It spawns multiple child processes to handle the load. However, these processes are standalone, meaning they initialize all modules (including mod_php) even for static file requests. This behavior results in high RAM usage as seen in figure 3.1.

mod_php is a PHP interpreter module for Apache. It is the most common way of executing PHP scripts when using Apache HTTP. The module is loaded in each Apache process, thus a request for a PHP script is executed directly within the process, not being relayed to another server with the ability to run the script. The main disadvantage to embedding PHP inside Apache is that if a user is requesting a static resource such as CSS, JS or an image, it still has to go through Apache process with PHP module, therefore increasing RAM and CPU usage.

3.1.1 Load testing Apache with mod_php

We are going to load test Apache with mod_php running a simple WordPress-powered site. Within your command line, navigate to the wordpress-ansible directory and run the ”apache_mod_php.yml” Ansible playbook:

```
ansible-playbook -i hosts apache_mod_php.yml
```

The playbook will configure Apache and create a virtual host for the WordPress site. We are using a default Apache HTTP configuration file without any performance modifications. [14]

To deploy the simple WordPress site, run the "wordpress_basic.yml" Ansible playbook. It will download the latest version of WordPress with the default theme and installs the database tables. We are now ready to carry out the testing.

At the loader.io load testing service, we have created a new test [15], configuring it to send from 0 to 200 simultaneous client requests to the index of our testing server for a duration of 30 seconds. Figure 3.1 depicts the final chart.

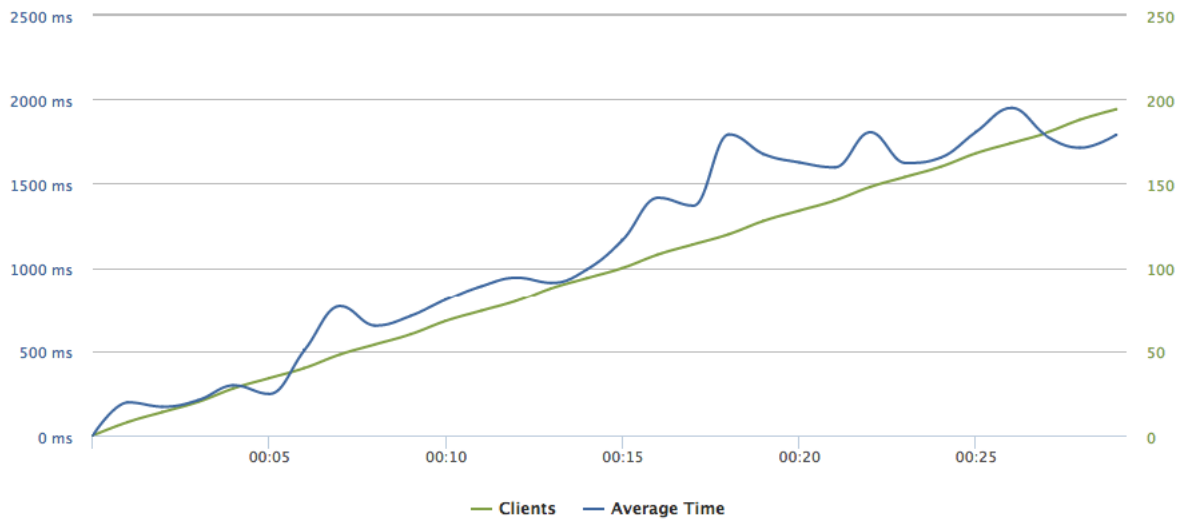

Figure 3.1: Apache HTTP with mod_php: clients versus average response time

This chart shows the average server response times when a number of simultaneous client requests are sent to it. We can observe that even under a high load of 200 concurrent requests the average response time is under 2 seconds. It grows approximately linearly.

However, to truly see how the server copes with the load, the figures 3.2 and 3.4 come in handy. They represent screenshots of Htop interactive process viewer [23] during 2 and 25 seconds into the load testing, respectively.

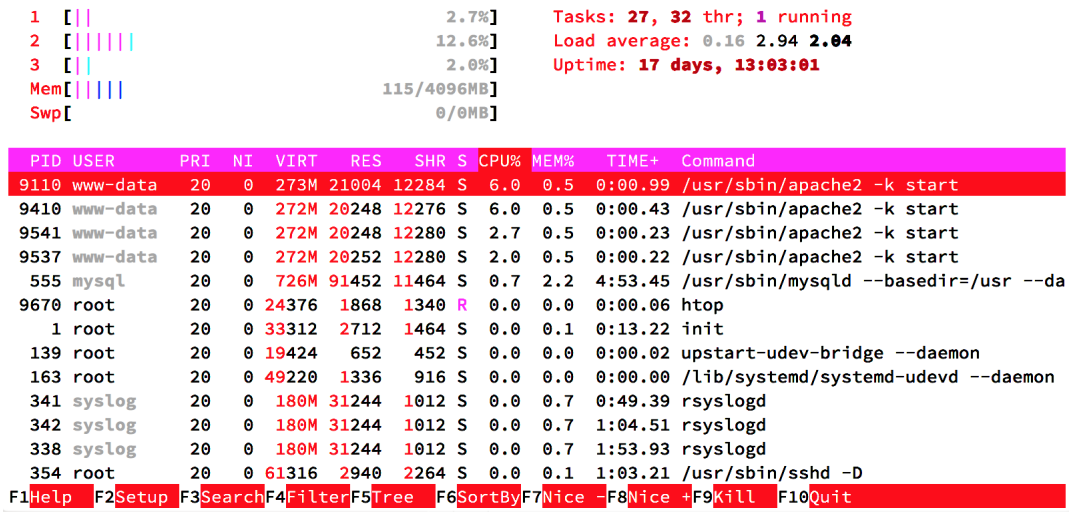

Figure 3.2: Apache HTTP with mod_php: Htop process viewer 2 seconds into test

In the screenshot, we can see a table of currently running processes. The columns RES, CPU% and MEM% show the RAM usage in kilobytes, CPU usage in percentage and RAM usage in percentage of a process, respectively. What is more, there are several gauges, visually depicting the usage or particular CPU core (or thread), RAM and Linux Swap. Opening the Htop's help screen (F1 in Htop), the gauges are explained as follows:

Figure 3.3: Screenshot of Htop's help screen explaining the main gauges

It is crucial to notice that the yellow bars in the Memory bar section show cached memory, which is not counted towards the used memory since it can be misleading at times (will discuss this a bit later).

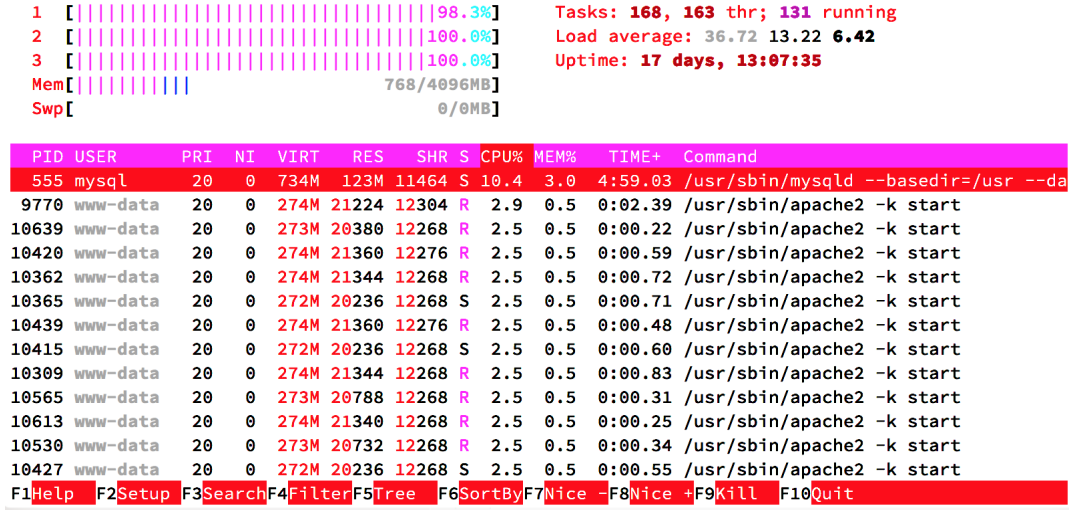

Figure 3.4: Apache HTTP with mod_php: Htop process viewer 25 seconds into test

From both screenshots, we can observe that Apache HTTP, represented by the process /usr/sbin/apache2 is, at first, just warming up, spawning three child processes and under heavy load, consuming the CPU completely and taking about 770 megabytes of memory. It does not tell us much without comparing it to the results of benchmarking different server stacks, what we are going to do shortly.

Loader.io testing results page [15] also shows us more statistics about the test:

- **Average response time:** 1229 ms
- **Min/Max response times:** 144 / 4660 ms
- **Count of successful responses:** 2178

3.2 Nginx with PHP-FPM

”Nginx (pronounced ”engine-x”) is an open source [...] web server. The nginx project started with a strong focus on high concurrency, high performance and low memory usage. [...] Nginx uses an asynchronous event-driven approach to handling requests, instead of the Apache HTTP Server model that defaults to a threaded or process-oriented approach, where the Event MPM is required for asynchronous processing. Nginx’s modular event-driven architecture can provide more predictable performance under high loads.” [42]

Due to Nginx’s asynchronous event-driven nature, its primary use case scenario is to relay the incoming requests to another application for processing, such as PHP-FPM, as quickly as possible, thus being able to handle large amounts of concurrent connections.

PHP-FPM is a FastCGI server bound to a TCP port or socket, bundled with the official PHP interpreter. It listens for PHP requests (request for a PHP script), responding with the resulting content (usually HTML). Comparing Apache’s mod_php with PHP-FPM, while a PHP request is processed by Apache directly, in case of Nginx, it has to be relayed to PHP-FPM, which is able to process it. We do not have to know how it works in a greater depth to understand that when a request is flowing through multiple applications, more server resources have to be consumed. On the other hand, as Nginx and PHP-FPM are specialized applications, they are more efficient when put together. It can be observed from the figures below.

3.2.1 Load testing Nginx with PHP-FPM

Within your command line, navigate to the wordpress-ansible directory and run the ”nginx_php-fpm.yml” Ansible playbook:

```
ansible-playbook -i hosts nginx_php-fpm.yml
```

Analogous to the load testing Apache with mod_php, we create a new test on Loader.io with the same configuration. [20]

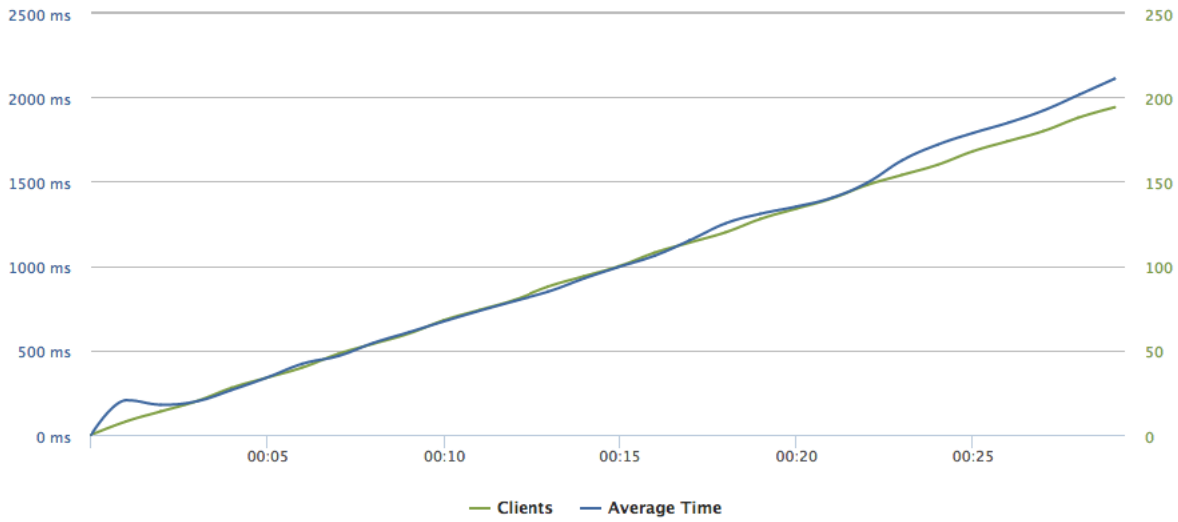

Figure 3.5: Nginx with PHP-FPM: clients versus average response time

Figure 3.5 shows us similar average response times to the figure 3.1 (Apache with mod_php), even slightly worse. However, screenshots of Htop during 1 and 22 seconds into the test depict a different situation. As we can see, even under 150 simultaneous client requests, server RAM usage is only at 46 MB. Compared with Apache’s 770 MB, by installing Nginx with PHP-FPM we optimized the server markedly. We can also see that Nginx is not using more than 3% of the server’s CPU for the reason that it was programmed to manage thousands of concurrent connections and we sent only 200 of them. A better benchmark for Nginx performance is described in the 4.2 section.

Although we could tweak Apache with mod_php to have a better performance, it has a steep learning curve, requiring a considerable amount of time. The objective of this work is to provide web administrators and WordPress developers with solutions they can easily implement, therefore we will not elaborate on this topic in greater detail.

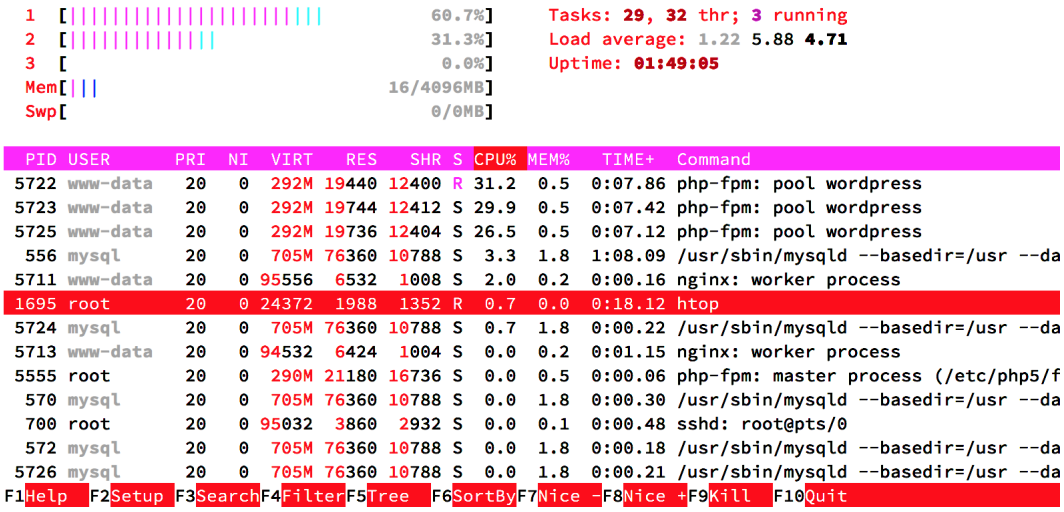

Figure 3.6: Nginx with PHP-FPM: Htop process viewer 1 second into test

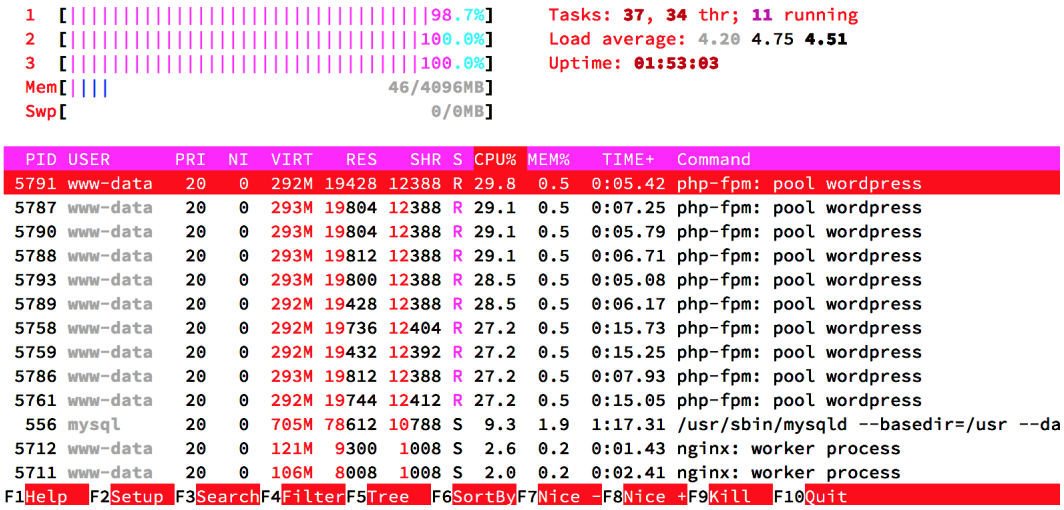

Figure 3.7: Nginx with PHP-FPM: Htop process viewer 22 seconds into test

Astute readers might notice that MariaDB process (/usr/sbin/mysqld) consumes more memory (RES column) than Htop's Mem gauge total used memory is showing (46 MB) alone. As we have explained from the figure 3.3, memory bar is not counting the cached memory to its total used value. The only indication is the yellow bars which follow the green ones.

3.3 Nginx + HHVM

In this section, we are going to substitute PHP-FPM with a different FastCGI PHP interpreter, HHVM and load-test it as we did with the earlier technologies.

”HipHop Virtual Machine (HHVM) is a process virtual machine based on just-in-time (JIT) compilation, serving as an execution engine for PHP. [...] By using the principle of JIT compilation, executed PHP [...] code is first transformed into intermediate HipHop bytecode (HHBC), which is then dynamically translated into the x86-64 machine code, optimized and natively executed. This contrasts to the PHP’s usual interpreted execution, in which the Zend Engine transforms the PHP source code into opcodes as a form of intermediate code, and executes the opcodes directly on the Zend Engine’s virtual CPU.”. [40]

3.3.1 Load testing Nginx with HHVM

Within your command line, navigate to the wordpress-ansible directory and run the ”nginx_hhvm.yml” Ansible playbook:

```
ansible-playbook -i hosts nginx_hhvm.yml
```

Analogous to the load testing Apache with mod_php, we create a new test on Loader.io with the same configuration. [17] As HHVM is transforming and optimizing the PHP code during the initial requests, we have made several ones before running the load testing to get more accurate results (response time for the initial request was more than 10 seconds long).

Figure 3.8: Nginx with HHVM: clients versus average response time

Reviewing the above chart, we can see that average response times have decreased from the 2-second to under 1-second levels. When 200 simultaneous requests are sent to the server, we receive them after 800 ms in average. This means that just by using HHVM instead of PHP-FPM, the performance of our server increases 2,5 times. What is more, the count of successful responses leaps to around 5800, thus nearly tripling the throughput of the server. [17]

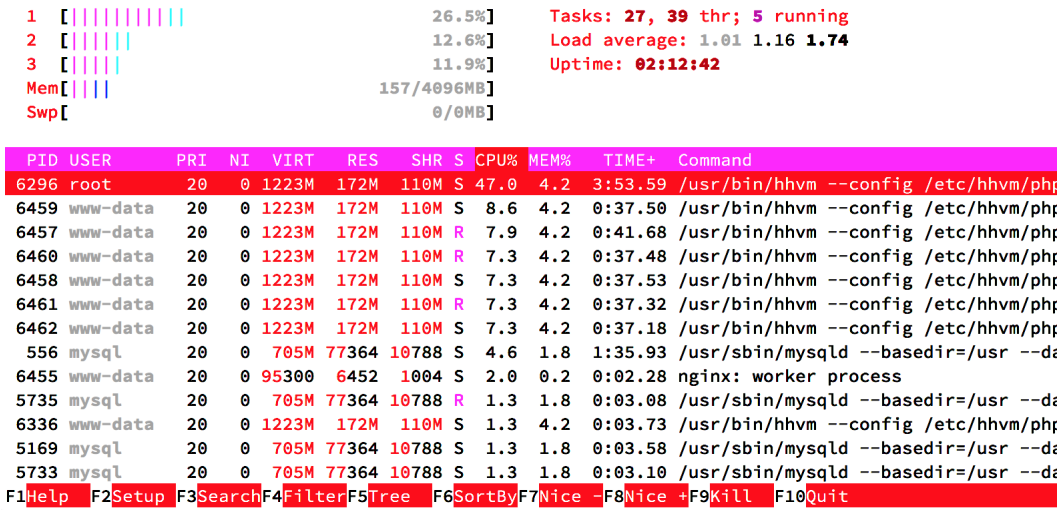

Figure 3.9: Nginx with HHVM: Htop process viewer 1 second into test

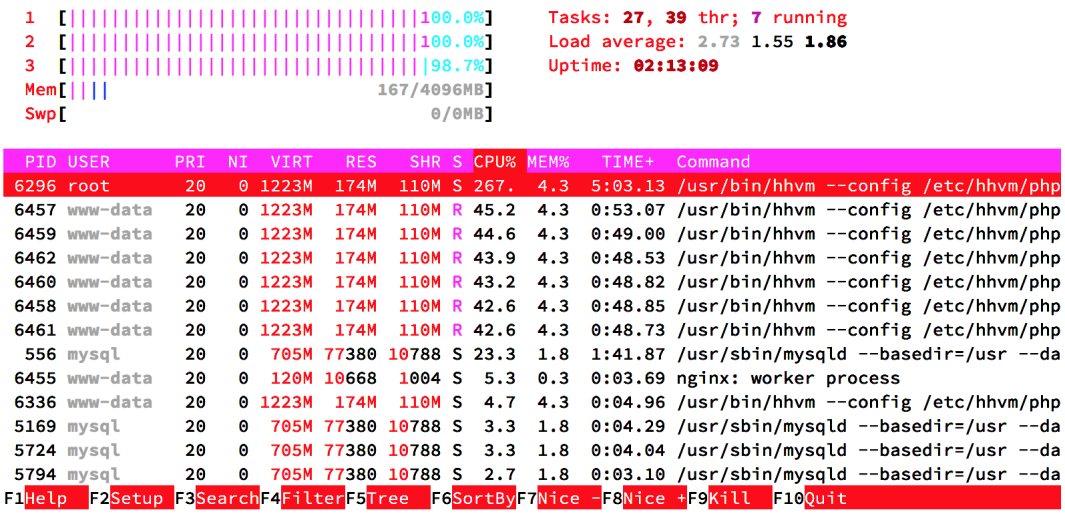

Figure 3.10: Nginx with HHVM: Htop process viewer 22 seconds into test

Looking at the Htop process viewer screenshot, in comparison to using PHP-FPM, RAM usage more than tripled. However, it is still relatively low, especially compared to the memory usage when using Apache with mod_php.

If this was a competition, choosing Nginx with HHVM as your web-serving stack would beat all its contestants, therefore we recommend going with it.

3.3.2 Advanced WordPress and HHVM

Our standard test did not benchmark HHVM's full potential, as we could observe from the figure 3.8. We will make the WordPress-powered site more complex. The new installation will include:

- several plugins, such as WooCommerce, Jetpack, WordPress SEO by Yoast and others [12]
- a free e-commerce theme Storefront [3]
- WooCommerce dummy data
- WP Test dummy data [31]

In order to have the advanced WordPress configured, run the ”wordpress_advanced.yml” playbook:

```
ansible-playbook -i hosts wordpress_advanced.yml
```

We configured the Loader.io test to send requests to two different URL addresses, the root of the server and to ”?product=woo-ninja-3” (a WooCommerce product page). [19]. The results can be seen in the figure 3.11. Average response times declined from under 1-second to around 7-second levels, which is also reflected in the count of successful responses (1237 versus 5769). Installing complex WordPress plugins and themes has clearly a profound impact on the site’s performance.

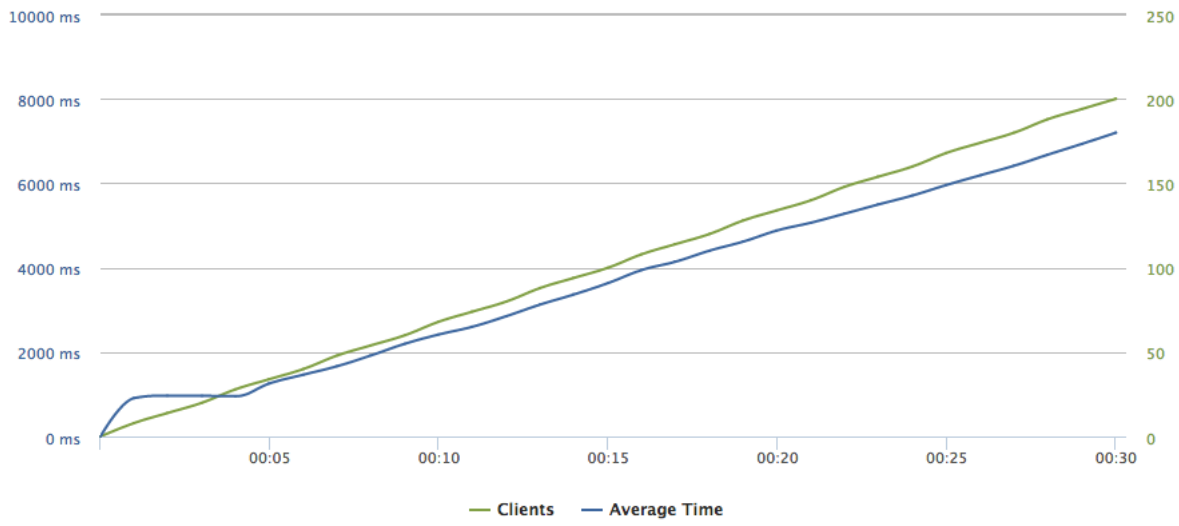

Figure 3.11: Nginx with HHVM and advanced WordPress: clients versus average response time

4. Caching

4.1 Database caching

WordPress was programmed with the support of database caching in mind. Core functions and methods querying the database (such as get_option, get_posts and others) use WordPress Object Cache API [48] on the background.

The default Object Cache API caches the database queries and results in a PHP array — only for a duration of the script execution (one request). On one hand, this mechanism is useful in templates, where the same information about the site is retrieved multiple times (for example get_bloginfo function). On the other hand, when many users visiting our site request the same page, the same database queries have to be made unnecessarily multiple times.

If we need to store the database cache persistently, we can use an in-memory database. ”An in-memory database is a database management system that primarily relies on main memory for computer data storage. It is contrasted with database management systems that employ a disk storage mechanism. Main memory databases are faster than disk-optimized databases since the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk.” [41]

The two most popular in-memory databases in the WordPress ecosystem are Memcached and Redis. In our work, we have chosen to use Redis as it is modern, well-supported and having all the features of Memcached. [56] In order for WordPress Object Cache API to use Redis for database caching, we need to install Redis server and Redis PHP connector (driver). We have prepared an Ansible playbook which downloads a PHP driver for Redis and installs it into the currently installed WordPress site (whether basic or advanced). Within your command line, navigate to the wordpress-ansible directory and run the ”wordpress_redis_db_cache.yml” Ansible playbook:

```
ansible-playbook -i hosts wordpress\_redis\_db\_cache.yml
```

Analogous to how we benchmarked the web-serving software in the previous chapter, we

configured a Loader.io test and performed several load tests for the site with Redis database caching enabled. Suprisingly, resulting charts did not show any noticeable improvements [18] over the other tests. Contemplating over this issue, we think that the database on our testing server is not a bottleneck. However, as this has not been proved yet, it might be a subject of a future work.

4.2 Page caching

Page caching is storing the output of a PHP interpreter (such as PHP-FPM or HHVM) on a disk or in RAM on the first request to a page (URL) and serving this stored (cached) content on subsequent ones. Page caching increases the performance of a web server considerably. To put it in simplistic terms, page caching converts a dynamic website into a static one. The most noticeable downside of page caching is that it is hard, if not impossible, to cache highly dynamic pages like constantly changing homepage or user profile page. In such circumstances, it is usually better to bypass the caching mechanisms and process the request with a PHP interpreter.

The easiest (but not the most performant) way to achieve page caching in a WordPress-based site is to use a plugin such as W3 Total Cache or WP Super Cache. These plugins store a page on your server's disk as an HTML file. When the page is requested for the second time, the HTML file is sent back as a response without having to process it within WordPress, thus saving hundreds to thousands of milliseconds. The largest advantages of these plugins are their ease of use. They usually come with simple user interface and a mechanism to bypass the cache for logged in users and highly dynamic pages. It is also quite straightforward to purge the cache if a new piece of content has been added with most of the plugins doing it automatically.

However, we prefer to set up page caching at a lower level, directly in Nginx. Nginx FastCGI module [25] includes a caching mechanism. The advantages of this approach are that the content is stored in RAM (as opposed to a disk) and with the use of various Nginx directives [24], we get a fine-grained control over the caching process. What is more, there is a plugin called Nginx Helper, which is able to clear the cache when a new WordPress post or page is added. In our work, we have used Nginx's FastCGI cache to increase the server's web-serving performance.

We are going to benchmark the performance of our testing server with Nginx FastCGI caching enabled. Create a new Loader.io test with a duration of 15 seconds, maintaining

client load from 300 to 1000 simultaneous clients to the root of our WordPress-powered site. [16] To configure the FastCGI caching, within your command line, navigate to the wordpress-ansible directory and run the ”nginx_hhvm_fastcgi_cache.yml” Ansible playbook:

```
ansible-playbook -i hosts nginx_hhvm_fastcgi_cache.yml
```

The resulting clients versus average response time chart can be observed in the figure 4.2. Note that we did a manual request (non-cached one) before starting the test for Nginx to cache it.

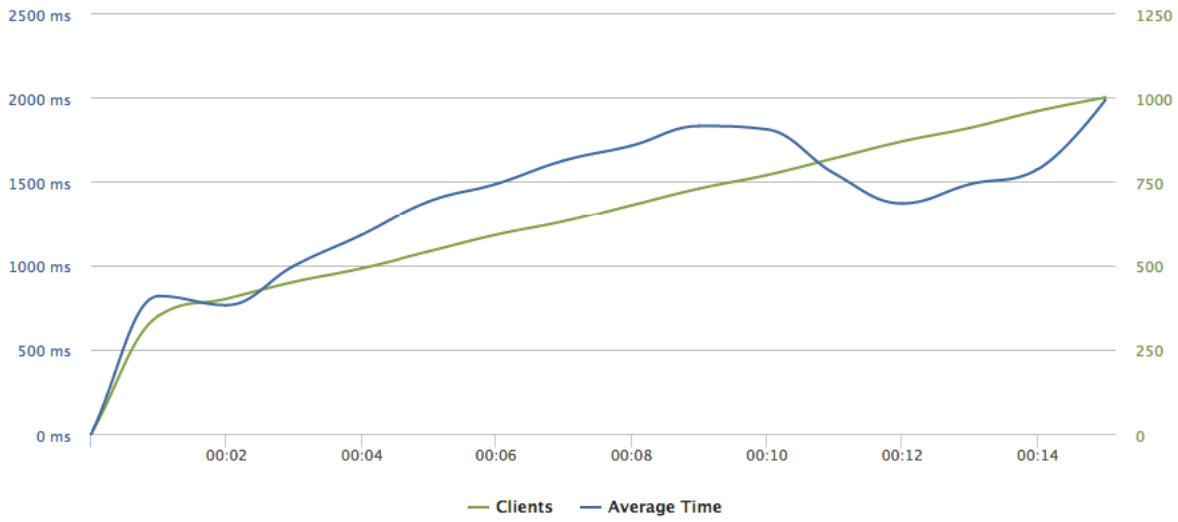

Figure 4.1: Nginx with FastCGI caching: clients versus average response time

With our three-CPU testing server, we were able to serve 1000 concurrent requests while maintaining the average response time under 2000 ms. Compare it with the previous testing **??**, where we barely served 200 concurrent requests under 8000 ms average response time. Another interesting statistic is the output of Htop process viewer, which can be observed in the figure below.

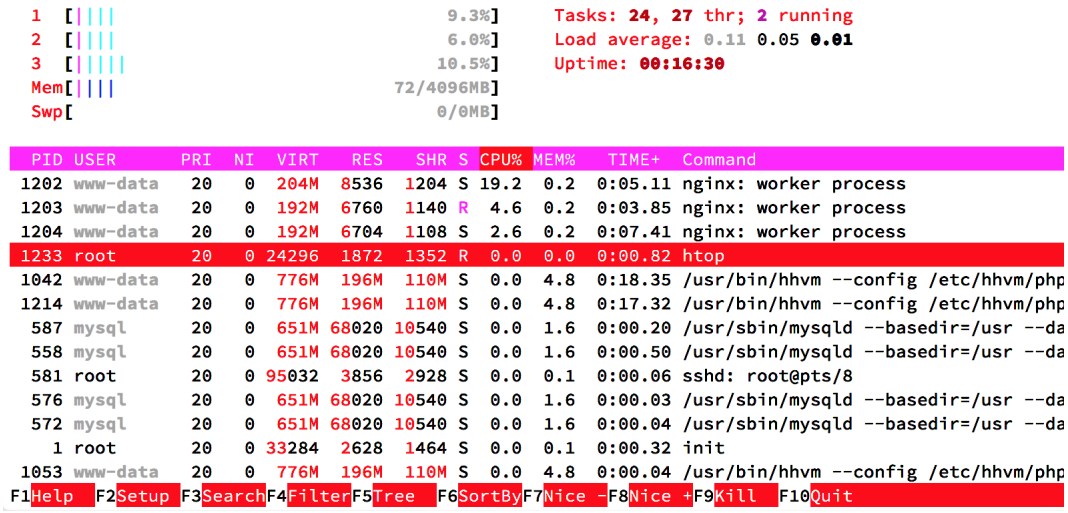

Figure 4.2: Nginx with FastCGI caching: Htop process viewer 9 seconds into test

9 seconds into the benchmarking (around 750 simultaneous requests), the server's CPU usage is at less than one-quarter. The single cached page is not taking large amounts of memory, as can be seen in RAM usage (72 MB out of 4096). We can confidently conclude this section by saying that, as was stated before, Nginx is an efficient web-serving software.

4.3 Browser caching

Browser caching (also referred to as client-side caching) is the process of storing data in the client's browser memory. If we store the resources (CSS, JS, images and fonts) and HTML pages in the client's browser cache, the browser doesn't have to load the resources from our servers, therefore saving time and bandwidth as well as server processing power. The main disadvantage of browser caching is that if a resource was changed, the cache needs to be cleared (revalidated) manually. Fortunately, the easiest and most straightforward method to revalidate browser cache is to rename the resource. User's browser will assume that it is an entirely new one, thus downloading and caching it again.

W3 Total Cache, a popular WordPress performance-optimizing plugin, comes with a browser caching feature. We can specify the amount of time after which the cached resource will expire and be re-downloaded from our site. The plugin will then generate a Nginx configuration file which has to be included in the site's Nginx server block [27] as the information about browser caching is sent in HTTP headers. If we also enable minification, we can clear the browser cache by clicking on the "Update media query strings" button in the W3 Total Cache administration dashboard. It works by either renaming the resource or appending a different string to the end of the resource's URL address (identifier), (after the

”?” — also called a query string [44]).

For testing purposes, we can either use browser’s developer tools or a web service such as GTmetrix. Opening the Developer Tools in Google Chrome browser and navigating to the Network tab, we can observe resources of the site being downloaded. If a resource has been cached in the browser, it will show as ”(from cache)” in the column ”Size”. After submitting a site to GTmetrix for a performance report, it will inform us about the state of browser caching (among other useful statistics) — which resources are and are not cached on our site.

5. Source code performance optimizations

Since we have compared different kinds of server stacks in the third chapter and discussed the usage of caching in the fourth one, we will now look into optimizing the performance of source code of a WordPress-based website. The first step is to detect the underperforming parts of the site (profiling). Then, we can begin to optimize them by employing various techniques and following best practices.

5.1 Profiling source code with XHProf

”In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler.” [43]

”XHProf is a hierarchical profiler for PHP. It reports function-level call counts and inclusive and exclusive metrics such as wall (elapsed) time, CPU time and memory usage. A function’s profile can be broken down by callers or callees. The raw data collection component is implemented in C as a PHP Zend extension called xhprof. XHProf has a simple HTML based user interface (written in PHP). The browser based UI for viewing profiler results makes it easy to view results or to share results with peers. A callgraph image view is also supported.” [6]

After installing the XHProf PHP extension and a GUI [21], we can easily identify the most CPU and memory hungry functions in plugins and themes. Then, we might either remove or begin to optimize them. Although there exist several WordPress profiling plugins such as Query Monitor or P3, they are not as exact and verbose as XHProf.

5.2 Source code optimization techniques and best practices

5.2.1 Autoloading PHP Classes

”Many developers writing object-oriented applications create one PHP source file per class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).” [33] Some programmers who are indifferent to the performance of their source code solve this problem by including all of the classes in one general file. The problem with this approach is that the classes which are not needed for a current code execution (for example requests to administration panel do not require classes used on the front-side of the website) are unnecessarily included every time a request is made, thus consuming more memory and CPU.

A proper solution is to make use of PHP’s autoloading features. ”You may define [...] a function which is automatically called in case you are trying to use a class/interface which hasn’t been defined yet. By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error.” [33]

5.2.2 Time complexity of PHP functions

”In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. [...] Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.” [45]

Using more efficient algorithms in plugins and themes, we might save tens to hundreds of milliseconds in our WordPress-based website execution time. Therefore, it is beneficial to have a basic knowledge of algorithms time complexity, especially the big O of native PHP functions. [9]

5.2.3 WordPress Plugin API

Plugin API, employed throughout the whole WordPress, makes it one of the most extensible PHP-based CMS. It is based on event-driven architecture (Observer design pattern), comprising hooks, actions and filters. ”Hooks are provided by WordPress to allow your

plugin to 'hook into' the rest of WordPress; that is, to call functions in your plugin at specific times, and thereby set your plugin in motion." [50] Actions and filters are two kinds of hooks.

"**Actions** are triggered by specific events that take place in WordPress, such as publishing a post, changing themes, or displaying administration screen. An Action is a custom PHP function defined in your plugin (or theme) and hooked, i.e. set to respond, to some of these events."

"**Filters** are functions that WordPress passes data through, at certain points in execution, just before taking some action with the data (such as adding it to the database or sending it to the browser screen). Filters sit between the database and the browser (when WordPress is generating pages), and between the browser and the database (when WordPress is adding new posts and comments to the database); most input and output in WordPress passes through at least one filter. WordPress does some filtering by default, and your plugin can add its own filtering."

The performance optimizations of our website are based on utilizing proper action and filter hooks for our functions. To give you an example, when a plugin or a theme is activated (or deactivated), special action hooks are triggered. We should use them to initialize a plugin or a theme (e.g. create new user role, add rewrite rules — operations inserting data into database) instead of running these operations on each website execution (each request).

5.2.4 WordPress database queries and structure

The database structure of a WordPress-based site is rather flexible, being able to accommodate various kinds of data. The "posts" table holds records of pages, posts and custom post types. [51] Table "postmeta" can store additional information (metadata) for particular entries from the "posts" table. "term_taxonomy" table is used for data classification (categorization), holding terms from different taxonomies. [52]

When adding data to posts, we can both add them as "postmeta" to a particular post, or create a new taxonomy with terms for the post. If we need to query the database by these additional data, it is better to add them as taxonomy with terms to obtain a greater querying performance. On the other hand, if the data are unique to a specific post and we do not generally need to query the database by it, it is better to add them as post's "postmeta" to save database disk space usage. [46]

As far as querying data from a WordPress database is concerned, there are several principles

we should follow to increase its performance: [1]

- Use WP_Query class for database queries.
- Only run the queries that you need.
- Do not query all the posts at once — paginate them instead.

When working with specific data which do not fit into default WordPress tables (having worse performance or for logical reasons), it is better to create a custom table. In that case, we should use the method ”prepare” of the ”wpdb” class, automatically instantiated as ”$wpdb” global variable to query these tables, as this method sanitizes user input, preventing SQL injection and other kinds of attacks.

6. Concluding remarks

7. Future work

7.1 Load balancing

Load balancing is a method of distributing computations across multiple servers, thus optimizing server use, maximizing throughput, minimizing response time and avoiding overload of a single server. Nginx's module called upstream [26] is used to define groups of servers that can be referenced in server blocks of different sites.

7.2 Perception of speed

Apart from optimizing the web server software and hardware efficiency, it is important to design a web application in a way that it will be perceived as fast and responsive by its visitors. One method of doing so is to use asynchronous JavaScript (Ajax) techniques to load additional data from a server after a site has been loaded by a user while providing visual guides of the to-be loaded data on the site.

References

[1] 10UP, *10up engineering best practices*, URL: https://10up.github.io/Engineering-Best-Practices/php/ (visited on 05/21/2015).

[2] ANSIBLE, *Installation — ansible documentation*, URL: http://docs.ansible.com/intro_installation.html (visited on 05/23/2015).

[3] AUTOMATTIC, *Storefront. the official woocommerce theme*, URL: http://www.woothemes.com/storefront/ (visited on 05/23/2015).

[4] CUTTS, Matt, *Google incorporating site speed in search rankings*, Apr. 9, 2010, URL: https://www.mattcutts.com/blog/site-speed/ (visited on 02/09/2015).

[5] DIGITALOCEAN, *Ssd cloud server, vps server, simple cloud hosting*, URL: https://www.digitalocean.com/ (visited on 05/23/2015).

[6] FACEBOOK, *Xhprof documentation — web archive*, URL: http://web.archive.org/web/20110514095512/http://mirror.facebook.net/facebook/xhprof/doc.html (visited on 05/23/2015).

[7] GARRON, Guillermo, *Wordpress performance comparison: using nginx, apache, apc and varnish in different scenarios*, Apr. 27, 2012, URL: http://www.garron.me/en/linux/apache-vs-nginx-php-fpm-varnish-apc-wordpress-performance.html (visited on 02/08/2015).

[8] GRIGORIK, Ilya, *Optimizing encoding and transfer size of text-based assets*, Apr. 1, 2014, URL: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer (visited on 05/23/2015).

[9] HOPKINS, Kendall, *Performance - list of big-o for php functions - stack overflow*, Apr. 20, 2010, URL: http://stackoverflow.com/questions/2473989/list-of-big-o-for-php-functions (visited on 05/23/2015).

[10] JACOB, Sherice, *Speed is a killer – why decreasing page load time can drastically increase conversions*, URL: https://blog.kissmetrics.com/speed-is-a-killer/ (visited on 01/15/2015).

[11] KEGEL, Dan, *The c10k problem*, URL: http://www.kegel.com/c10k.html (visited on 05/23/2015).

[12] LAMOS, Rastislav, *Ansible group variables file on wordpress-ansible github repository*, URL: https://github.com/lamosty/wordpress-ansible/blob/master/group_vars/webservers (visited on 05/23/2015).

[13] —, *Ansible playbooks for installing multiple kinds of server stacks with performance optimizations and security in mind.* URL: https://github.com/lamosty/wordpress-ansible (visited on 05/23/2015).

[14] —, *Apache http server default configuration file*, URL: https://github.com/lamosty/wordpress-ansible/blob/master/roles/apache/templates/apache2.conf.j2 (visited on 04/15/2015).

[15] —, *Apache2 mod_php loader.io testing results page*, URL: http://ldr.io/1IQVNXz (visited on 04/15/2015).

[16] —, *Nginx hhvm advanced fastcgi caching: load testing with loader.io*, URL: http://ldr.io/1cI1XLY (visited on 05/23/2015).

[17] —, *Nginx hhvm loader.io testing results page*, URL: http://ldr.io/1cbbomp (visited on 05/04/2015).

[18] —, *Nginx hhvm redis db caching: a load test by loader.io*, URL: http://ldr.io/1cbcY7Y (visited on 05/23/2015).

[19] —, *Nginx hhvm with advanced wordpress loader.io testing results page*, URL: http://ldr.io/1IQW6le (visited on 05/04/2015).

[20] —, *Nginx php-fpm loader.io testing results page*, URL: http://ldr.io/1cT2D14 (visited on 05/04/2015).

[21] —, *Profiling wordpress with xhprof on mac os x 10.10 - wordpress explained*, Mar. 19, 2015, URL: https://lamosty.com/2015/03/profiling-wordpress-with-xhprof-on-mac-os-x-10-10/ (visited on 05/23/2015).

[22] LOADER.IO, *Application load testing tools for api endpoints with loader.io*, URL: http://loader.io/s/XoYdj (visited on 04/15/2015).

[23] MUHAMMAD, Hisham, *Htop — an interactive process viewer for linux*, URL: http://hisham.hm/htop/ (visited on 04/15/2015).

[24] NGINX, *Module ngx_http_core_module*, URL: http://nginx.org/en/docs/http/ngx_http_core_module.html (visited on 04/18/2015).

[25] —, *Module ngx_http_fastcgi_module*, URL: http://nginx.org/en/docs/http/ngx_http_fastcgi_module.html (visited on 04/18/2015).

[26] —, *Module ngx_http_upstream_module*, URL: http://nginx.org/en/docs/http/ngx_http_upstream_module.html (visited on 05/23/2015).

[27] NGINX-COMMUNITY, *Serverblockexample - nginx community*, URL: http://wiki.nginx.org/ServerBlockExample (visited on 04/18/2015).

[28] NICHOLSON, Jacob, *Speed up php with apc - alternative php cache*, July 4, 2014, URL: http://www.inmotionhosting.com/support/website/what-is/speed-up-php-with-apc (visited on 02/09/2015).

[29] NOSILVER4U, *Ewww image optimizer*, URL: https://wordpress.org/plugins/ewww-image-optimizer/ (visited on 05/07/2015).

[30] NOTTINGHAM, Mark, *Caching tutorial for web authors and webmasters*, URL: https://www.mnot.net/cache_docs/ (visited on 02/10/2015).

[31] NOVOTNY, Michael, *Wp test — the best tests for wordpress*, URL: http://wptest.io/ (visited on 05/23/2015).

[32] OEZI, *Which browsers handle 'content-encoding: gzip' and which of them has any special requirements on encodinq quality?*, Nov. 19, 2011, URL: http://webmasters.stackexchange.com/questions/22217/which-browsers-handle-content-encoding-gzip-and-which-of-them-has-any-special (visited on 05/23/2015).

[33] PHP.NET, *Php: autoloading classes - manual*, URL: http://php.net/manual/en/language.oop5.autoload.php (visited on 05/23/2015).

[34] PUIG, Tomas, *Announcing wp engine's high availability hhvm platform: mercury – labs alpha*, Nov. 19, 2014, URL: http://wpengine.com/2014/11/19/hhvm-project-mercury/ (visited on 02/08/2015).

[35] ROBERTS, Harry, *Front-end performance for web designers and front-end developers*, Jan. 20, 2013, URL: http://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/ (visited on 02/10/2015).

[36] VPSFREE.CZ, *Vpsfree.cz - virtuální privátní servery svobodně*, URL: https://vpsfree.cz (visited on 05/23/2015).

[37] W3TECHS, *Usage statistics and market share of apache for websites*, URL: http://w3techs.com/technologies/details/ws-apache/all/all (visited on 05/23/2015).

[38] —, *Usage statistics and market share of wordpress for websites*, URL: http://w3techs.com/technologies/details/cm-wordpress/all/all (visited on 05/23/2015).

[39] WIKIPEDIA, *Apache http server*, URL: http://en.wikipedia.org/wiki/Apache_HTTP_Server (visited on 05/23/2015).

[40] —, *Hiphop virtual machine*, URL: http://en.wikipedia.org/wiki/HipHop_Virtual_Machine (visited on 04/15/2015).

[41] —, *In-memory database*, URL: http://en.wikipedia.org/wiki/In-memory_database (visited on 05/23/2015).

[42] —, *Nginx*, URL: http://en.wikipedia.org/wiki/Nginx (visited on 04/15/2015).

[43] —, *Profiling (computer programming) - wikipedia, the free encyclopedia*, URL: http://en.wikipedia.org/wiki/Profiling_(computer_programming) (visited on 05/23/2015).

[44] —, *Query string - wikipedia, the free encyclopedia*, URL: http://en.wikipedia.org/wiki/Query_string (visited on 05/23/2015).

[45] —, *Time complexity*, URL: http://en.wikipedia.org/wiki/Time_complexity (visited on 05/23/2015).

[46] WOOD, Samuel, *When to (not) use a custom taxonomy » otto on wordpress*, May 4, 2011, URL: http://ottopress.com/2011/when-to-not-use-a-custom-taxonomy/ (visited on 05/21/2015).

[47] WORDPRESS, *Blog tool, publishing platform, and cms*, URL: https://wordpress.org/ (visited on 05/23/2015).

[48] WORDPRESS-CODEX, *Class reference/wp object cache*, URL: https://codex.wordpress.org/Class_Reference/WP_Object_Cache (visited on 05/23/2015).

[49] —, *History*, URL: http://codex.wordpress.org/History (visited on 05/23/2015).

[50] —, *Plugin api*, URL: https://codex.wordpress.org/Plugin_API (visited on 05/21/2015).

[51] —, *Post types*, URL: https://codex.wordpress.org/Post_Types (visited on 05/21/2015).

[52] —, *Taxonomies*, URL: https://codex.wordpress.org/Taxonomies (visited on 05/21/2015).

[53] WORK, Sean, *How loading time affects your bottom line*, URL: https://blog.kissmetrics.com/loading-time/ (visited on 02/09/2015).

[54] WORTHAM, Steve, *Get number of concurrent requests by browser*, Sept. 17, 2011, URL: http://stackoverflow.com/questions/7456325/get-number-of-concurrent-requests-by-browser (visited on 02/09/2015).

[55] YU, Xiao, *Wordpress performance with hhvm*, Nov. 2014, URL: http://www.xyu.io/2014/09/wordpress-performance-with-hhvm/ (visited on 12/10/2014).

[56] ZULAUF, Carl, *Caching - memcached vs. redis? - stack overflow*, June 29, 2012, URL: http://stackoverflow.com/questions/10558465/memcached-vs-redis (visited on 05/23/2015).