

Tartu University
Faculty of Science and Technology
Institute of Technology

Charl Kivioja

Efficient caching of web content

Bachelor thesis (12 EAP)
Computer Engineering

Supervisors:

Jefims Gasels, B.Sc
Ahti Saar, M.Sc

Tartu 2023

Resümee/Abstract

Veebilehtede sisu efektiivne puhverdamine

Bakalaureusetöö eesmärk oli arendada kohandatud vahemälulahendus sisuhaldussüsteemide vaatenurgast ning võrrelda seda kahe turul saadavaloleva tootega - Nginx ja Varnish. Uuringu tulemused näitavad, et kohandatud lahendus on kiiruse poolest parem kui mõlemad tooted ning seda võib kasutada alusena uute ja tõhusamate sisuhaldussüsteemide loomiseks. Arendatud lahendus kasutab rohkem mälu ja vajab rohkem protsessoriressursse. Nginx pakkus aeglasemat kiirust, kuid stabiilset jõudlust koormuse all. Bakalaureusetöö tulemused on olulised tulevaste CDN-lahenduste arendamisel ning võimaldavad valida parima vahemälulahenduse.

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Märksõnad: Rust, sisuedastusvõrk, RocksDB, puhverserver, jõudlus, reeper

Efficient caching of web content

The objective of the bachelor's thesis was to develop a custom caching solution from the perspective of CDN servers and compare it with two available products in the market - Nginx and Varnish. The study results indicate that the custom solution is faster than both products and can be used to create new and more efficient CDN servers. However, the developed solution uses more memory and requires more processor resources. On the other hand, Nginx offered slower speed but stable performance under load. Therefore, the bachelor's thesis results are important for developing future CDN solutions and enabling the selection of the best caching solution.

CERCS: P170 Computer science, numerical analysis, systems, control

Keywords: Rust, Content Delivery Network, RocksDB, cache server, performance, benchmarking

Contents

Resümee/Abstract.....	2
List of Figures.....	5
List of Tables	6
Acronyms	7
1 Introduction	8
2 Background	10
2.1 CDN	10
2.2 Rust programming language	11
3 Methodology	13
3.1 Setting up Nginx as a reverse proxy caching Server	14
3.2 Setting up Varnish as a reverse proxy caching Server.....	14
3.3 Setting up a virtual machine for Rust Deployment.....	15
3.4 Rust Development.....	16
3.4.1 RocksDB library.....	16
3.4.2 Additional Rust Libraries	16
3.4.3 Development of the caching solution.....	17
3.5 Benchmarking.....	18
4 The Results	21
5 Conclusion.....	28
Bibliography	30
Appendices	33
1 GitHub repository links for the projects	33
Licence	34

List of Figures

<i>Figure 1. Bash script used for server resource monitoring</i>	<i>18</i>
<i>Figure 2. Part of the Windows Powershell script was used to test the Rust server.....</i>	<i>19</i>
<i>Figure 3. Windows Powershell script used to monitor resource usage</i>	<i>20</i>
<i>Figure 4. 500KB file size speed comparison under load</i>	<i>22</i>
<i>Figure 5. 1MB file size speed comparison under load</i>	<i>23</i>
<i>Figure 6. 5MB file size speed comparison under load</i>	<i>23</i>
<i>Figure 7. 10MB file size speed comparison under load</i>	<i>23</i>
<i>Figure 8. 20MB file size speed comparison under load</i>	<i>24</i>
<i>Figure 9. 50MB file size speed comparison under load</i>	<i>24</i>
<i>Figure 10. System RAM usage comparison during testing.....</i>	<i>25</i>
<i>Figure 11. System CPU usage comparison during testing</i>	<i>26</i>

List of Tables

<i>Table 1. Averaged results across all test suites</i>	<i>21</i>
---	-----------

Acronyms

CDN – Content Delivery Network

ISP – Internet Service Provider

HTTP – Hypertext Transfer Protocol

IP – Internet Protocol

URL – Unified Resource Locator

ETAIS – Estonian Scientific Computing Infrastructure

HPC – High-Performance Computing Center

GB – Gigabyte

KB – Kilobyte

MB – Megabyte

RAM – Random Access Memory

SSH – Secure Shell

API – Application Programming Interface

CPU – Central Processing Unit

GNU – GNU's Not Unix

CSV – Comma Separated Value

1 Introduction

Latency is a significant challenge that confronts internet traffic. Despite the efforts to increase bandwidth and upgrade ISP speeds, these measures still need to be improved in accommodating the growth of data transfer rates. As a result, loading web pages often takes several seconds, a process called content delivery. Cisco has predicted that global IP traffic will reach an annual rate of 4.8 zettabytes per year by 2022, representing an 11-fold increase from 2012, with a further expected rise. These numbers are bound to increase the latency of content delivery even further [1] [2] [3].

Latency is attributable to the distance a web request must travel on the internet before it reaches its destination and can start returning with a result. Moreover, when numerous devices access the internet simultaneously, there is a high probability that the same web resource is being accessed concurrently from multiple locations worldwide, causing additional latency. This results from the fact that the web server must simultaneously serve content to various devices.

One possible approach to tackling this problem is to deploy Content Delivery Networks (CDNs) at various geographic locations closer to the user than the actual server. CDNs function as caches for the content to be served, checking their cache for the requested content when a request reaches the server. If the cache contains the query result, the CDN sends it back to the user; otherwise, it sends the request to the original server to be handled. CDNs help reduce the load on the original server and achieve lower latency by being physically closer to the end user, requiring fewer network hops to reach them [4].

CDNs are now a ubiquitous feature of the internet. In 2022, Cisco reported that over half (56%) of all internet traffic went through a CDN, and this trend is expected to continue growing. Some of the most widely used CDN server software include Nginx and Varnish, free-to-use software that can set up a caching HTTP reverse proxy for storing web content. Varnish is highly customisable, while Nginx is newer and faster. The main feature that the free version of Nginx lacks, however, is the selective purging of its cache [5] [6] [3].

This thesis aims to develop an optimal caching solution that can compete with Nginx and Varnish regarding performance. The solution will be implemented as a Rust HTTP server with a RocksDB embedded database for fast content caching. The custom caching solution will be benchmarked against Nginx and Varnish installations to assess the viability of this approach

compared to commercial products. The final stage of the thesis will involve analysing the developed custom caching solution, and the results can be used in building future CDN solutions. [7] [8].

The thesis is structured into five chapters, each serving a specific purpose in addressing the research question. Chapter 1 introduces the research topic, outlines the research question, and describes the significance of the study. Chapter 2 reviews the relevant theoretical background in the field. Chapter 3 describes the methodology used in the study, including data collection and analysis techniques. Chapter 4 presents the study's findings, including any analyses and visual representations of the data. Chapter 4 also discusses the findings, relating them to the research question and addressing any limitations or implications of the study. Finally, Chapter 5 concludes the study, summarising the main findings and suggesting avenues for future research.

2 Background

A caching server is a web server that stores frequently accessed data in memory or on disk to reduce the response time of subsequent requests. Rust is a programming language known for its high performance and memory safety, and RocksDB is an embedded key-value database library optimised for flash storage [9] [10].

Implementing a caching server using Rust and RocksDB offers several advantages over other programming languages and storage solutions. Rust's ownership and borrowing system ensure that memory management is efficient and safe, reducing the likelihood of memory leaks and null pointer errors. RocksDB's efficient compression and memory mapping capabilities ensure data is stored and accessed quickly and efficiently [9] [10].

2.1 CDN

Content Delivery Networks (CDNs) serve to cache specific static elements from an origin server closer to the end user. This approach offers reduced latency and lower network bandwidth consumption from the end user's perspective. The CDN caches content on a request basis and begins by examining its cache to determine whether the requested content is already present. The CDN forwards the request to the origin server if the cache does not contain the requested content. The origin server responds to the request, which the CDN caches before returning it to the user. The ideal CDN goal is to achieve a 100% cache hit ratio, implying that every incoming request corresponds to the related content ready for the user in its cache. CDN servers may also provide additional features, including pre-loading files into the cache, cache clearing, and modifying cache retention times. As CDNs are not reliant on the origin server, they can cache content from multiple web pages simultaneously. This approach is preferable for reducing network congestion at the edges of Internet Service Provider (ISP) networks or other large, interconnected networks. Although increasing network bandwidth and reducing latency are other potential solutions, they are typically expensive and time-consuming. As a result, CDNs are highly preferred for resolving these issues [1] [11] [12].

While CDNs are generally known to improve content delivery speed, they can also have potential downsides. One such disadvantage is the added complexity of the web architecture. Since the CDN sits between the origin server and the end user, additional network hops are

introduced, which may increase latency and negatively impact performance if not correctly configured. Another potential disadvantage of CDNs is stale content, where users may receive outdated content in the CDN cache. To mitigate these risks, CDNs implement a variety of mechanisms, such as cache control headers and cache invalidation, to ensure that content is up-to-date and delivered efficiently. It is also important to note that CDNs may not be suitable for all web applications, and their effectiveness may depend on various factors, such as the size and nature of the content being delivered and the geographical location of end-users [1] [11] [12].

2.2 Rust programming language

Rust is a compiled programming language that prioritises speed in terms of code performance and development speed. Intending to be as performant as other lower-level languages such as C, Rust has no garbage collector or functions for managing memory. Instead, Rust uses the concepts of ownership and borrowing to manage memory. Ownership defines the principles for how Rust manages memory, with every variable in Rust being immutable by default. By storing these variables in stack memory, Rust minimises performance overhead during execution.

In contrast, every mutable value is stored in heap memory. Furthermore, the concept of borrowing allows for passing values through references, with references ensuring that the value they point to exists for the variable's lifetime. Finally, rust performs memory safety checks during compile time to identify and rectify any potential mistakes in memory management [7] [10].

Rust is suitable for developing a caching solution due to its performance, memory safety, and concurrency features. Rust's performance results from its ability to compile to machine code, making it capable of running with low overheads. Furthermore, Rust's ownership and borrowing concepts ensure memory safety. They allow the compiler to check at compile time that there are no violations in memory management, preventing common errors such as null pointer dereferences, buffer overflows, and use-after-free vulnerabilities. This characteristic makes Rust a suitable choice for implementing a caching solution, as it can help to prevent issues such as stale content and cache invalidation [7] [10].

In addition, Rust's support for concurrency is another crucial aspect that makes it an excellent choice for developing a caching solution. By utilising Rust's concurrency features, developers can write thread-safe code that runs efficiently on multi-core processors, which is essential for a caching server. As a result, Rust's combination of performance, memory safety, and

concurrency features make it an excellent option for building a custom caching solution [7]
[10].

3 Methodology

This study utilised the ChatGPT language model, which was trained using the GPT-3.5 architecture developed by OpenAI, to rephrase specific sentences initially written in informal language. This allowed conveying the same information more formally, improving the written work's overall quality and professionalism. Also, this thesis used Grammarly, an automated proofreading and grammar-checking tool, to check for spelling and grammatical errors [13] [14].

This study's initial step entailed setting up Varnish and Nginx caching servers. Subsequently, the performance of these servers was benchmarked using Apache benchmark, a tool that provides statistics about HTTP or HTTPS requests [15]. The primary objective of the initial benchmarking was to demonstrate that Nginx is faster than Varnish. Following this, the development of the Rust and RocksDB-based backend was initiated.

All caching servers were established in ETAIS, a cloud self-service portal provided by the Estonian Scientific Computing Infrastructures [16]. To ensure an accurate benchmarking result, virtual machines hosting the caching servers were equipped with the same specifications, including 2GB of RAM, one virtual CPU, and 20GB of storage. CentOS 7 was selected as the virtual machine operating system based on the author's prior experience. In addition, security groups were added to facilitate access to virtual machines. In the self-service portal, the security groups are defined as a ruleset that outlines network resource access, primarily opening ports and defining allowed protocols on VM-s, allowing access over the network. The caching servers were configured for three primary security groups, including the SSH security group that permits secure shell or SSH access to the virtual machine, requiring the opening of port 22 on the VM. In addition, the Ping Security Group allows ICMP network protocol access to virtual machines on any port, permitting the pinging of virtual machines for debugging purposes. Finally, the Web Security Group defines access for TCP protocol on ports 80 for HTTP connections and 443 for HTTPS.

Ansible, a tool suite developed by Red Hat Software for automating tasks on Linux operating systems, including configuration management and deployment, was utilised to configure and deploy the servers. This choice was influenced by the features of Ansible, enabling ease of deployment and configuration management, making it an ideal tool for this study [17].

3.1 Setting up Nginx as a reverse proxy caching Server

Several packages were installed to configure the Nginx cache server, including Nginx, httpd, sysstat, and iotop, via the default package manager on CentOS 7 - yum. The EPEL-release repository provides additional packages for Enterprise Linux versions, including CentOS 7, giving access to the thNginx package from the yum package manager [18]. In addition, the httpd package was used to set up an Apache web server that will be used as a remote server, to have Nginx receive a request, forward it to the local Apache sample web server, and cache the result that the server sends back [19]. Finally, the sysstat and iotop packages provided tools for monitoring resource usage during benchmarking.

Custom modifications were needed to use Nginx as a caching server, starting with disabling the default Nginx web page by deleting the "server" code block from the main Nginx configuration file. The caching implementation involved defining a new "server" context block in the Nginx configuration and using the proxy_cache_path directive in the "http" context to define the location of the cache on the filesystem [20]. The new "server" context was set with the upstream proxy server at 127.0.0.1:8080, where the Apache web server listens. Nginx was then configured to use the previously defined cache to cache the results from this specific upstream server [21].

To benchmark the Apache web server, files of varying sizes (500KB, 1MB, 5MB, 10MB, 20MB, and 50MB) were populated, providing a comprehensive understanding of the capabilities of different caching software under various loads.

3.2 Setting up Varnish as a reverse proxy caching Server

Multiple packages were installed on the virtual machine to set up the Varnish caching server. The httpd package established an Apache web server as a remote backend for Varnish to forward requests to [19]. The EPEL-release repository was utilised to access additional packages required for installing Varnish on the system [18]. Pygpgme and yum-utils were installed as dependencies for Varnish. Pygpgme is a Python module focusing primarily on OpenPGP-based signing, verification, and encryption tasks [22]. Finally, sysstat and iotop packages were installed to enable resource monitoring during benchmarking.

Editing the httpd main configuration file was required to move the Apache web server from port 80 to port 8080. Once the Apache web server had been populated with different-sized files, the Varnish installation could proceed.

Since the Varnish package is in a custom repository, the yum package manager cache must be

updated with the repository to allow it to search for and install Varnish packages. After updating the cache, Varnish can be installed using the yum package manager [23].

To configure the Varnish runtime properties, the systemd unit file for the Varnish process must be updated [23]. Port and cache configurations must be added as arguments to the Varnish binary. Varnish was configured to listen on port 80 with a 1GB hard disk cache enabled. The upstream server must then be specified, the Apache web server listening locally on port 8080, with the upstream server address set to 127.0.0.1:8080 [23].

The final step involves populating the Apache web server with the files used for benchmarking and copying the resource monitoring script to the virtual machine.

3.3 Setting up a virtual machine for Rust Deployment

The implementation of the caching server is custom-made and implemented in the Rust programming language. Hence, the virtual machine designated to deploy the server must support the building and compiling of Rust applications. The installation of Rust requires the download and execution of Rustup, which is the official Rust installer and version management tool available on the Rust language homepage [24]. Cargo, the build tool and package manager included with Rust, is utilised to build and run the Rust project, including the server's source code on the virtual machine.

After Rust's installation, the Rust project's source code is transferred to the virtual machine, which is achieved using a GitHub repository. Git is installed on the virtual machine with the help of the package manager yum. Subsequently, the source code is cloned to the virtual machine by executing the Git command "clone".

Additionally, to monitor system resources, installing sysstat and iotop is necessary. Compiling the dependent library, RocksDB, on the virtual machine requires installing C++ development tools. First, the "Development Tools" package group is installed using the yum package manager, which includes numerous packages for developing on Linux platforms. To compile the RocksDB library on CentOS 7, which requires support for C++ 17, it is necessary to install additional packages on the virtual machine. Next, the EPEL-release repository was added, granting access to extra Enterprise Linux packages. Following that, yum-utils and centos-release-scl packages were installed on the system [25] [26] [27] [28].

To initiate the Rust server on the virtual machine, a systemd service for the program is created. In Linux, services are defined by Unit files and managed by the systemd system and service manager [29]. The unit file is configured to run a predefined shell script. Inside the shell script,

the `centos-release-scl` package group activates newer versions of GCC and Clang. The system then navigates to the folder where the source code resides, pulls the latest version from Github, and uses the Rust build tool Cargo to build the project from the source code and run it.

3.4 Rust Development

3.4.1 RocksDB library

RocksDB is a key/value storage engine that Facebook developed. It is built entirely in C++ and includes an officially supported wrapper for Java. The engine stores keys and values as arbitrary byte streams, and it is based on LevelDB, providing backwards-compatible support with its APIs [9].

The primary design goals of RocksDB are to provide fast storage, efficient point lookups, and range scans. It is optimised for high random-read and high update workloads, which makes it well-suited for server-side applications [9]. RocksDB aims to be a server-side database, removing the need to query a database over the network. The main advantage of this approach is that it minimises the time spent waiting for network calls to resolve, thus making the server and database communication speed depend solely on the speed of the database itself [9].

RocksDB is built on a log-structured merge-tree (LSM-tree) data structure optimised for fast writes and efficient reads. This data structure uses a sequence of immutable data files, called SST files, that are merged periodically. The merged files form a larger SST file that is more efficient to read, improving read performance [9].

RocksDB is flexible regarding the data types it supports and the operations it provides. For example, it allows users to store arbitrary byte streams as keys and values, making it suitable for many use cases. Additionally, it offers various operations, such as get, put, delete, and batch, which are essential for a caching database [9].

3.4.2 Additional Rust Libraries

Using external libraries is a common practice in software development, as it allows developers to leverage existing code to accelerate the development process and improve the quality of the final product. For example, in the case of the caching solution developed for the CDN server, several external libraries were employed to support critical functionality.

The `hyper` library, for example, was used as an HTTP implementation library for Rust, providing support for HTTP/1 and HTTP/2 protocols with an asynchronous design. In addition,

its client and server API facilitated the implementation of a low-level caching solution with minimal overhead. This is particularly useful in the context of CDN servers, where speed and efficiency are paramount [30].

Another critical library used in developing the caching solution was Tokio, which provides tools for working with asynchronous tasks and applications. Its runtime functionality made it easy to run asynchronous code, avoiding potential blocking issues behind specific requests. Additionally, Tokio supports asynchronous I/O, including TCP and UDP sockets, and synchronisation primitives and channels, such as timeouts, sleep, and intervals, making it a valuable asset in developing the caching solution [31].

Finally, the `async_stream` library was utilised to create asynchronous streams of elements, enabling the streaming of values directly over the network to and from the database. This feature represents a significant advantage in terms of performance and speed for the caching solution, as it minimises the time spent waiting for network calls to resolve [32].

3.4.3 Development of the caching solution

Rust provides various development tools and features to assist programmers in their work. One such tool is the Cargo package manager, which automatically searches and downloads a project's dependencies based on definitions in the "Cargo.toml" file. The build tool parses and processes this file to obtain the required dependencies [10].

A basic HTTP server was established using the Tokio runtime and `hyper` to initiate the development process. `Hyper`'s documentation provided an excellent starting point with sample code later used to build upon. The server served the cached files by removing the URL's resource part from the request, later referred to as the key for the value stored in the database. To demonstrate the concept, a default RocksDB database connection was created [30] [9].

As this initial approach focused on testing the cache querying capabilities, data was manually loaded into the database on application start-up. Once the database connection was established, selected files from the file system were placed into the database for initial data loading. Subsequently, the only step remaining was to query the server with the correct filename, and the corresponding file would be served to the client.

The `get_pinned` method from the RocksDB library was utilised to optimise performance when reading values from the database, which returns a `DBPinnableSlice` object. This method reduces the memory copying overhead associated with the standard `get` method, which always results in at least one copy operation from the source to the target string value. For example,

suppose the source value is already in the block cache. In that case, the copying can be avoided using the `get_pinned` method, which returns the `DBPinnableSlice` object that wraps the primary `Slice` object and stores a pointer to the value in the block cache. This can be especially beneficial when dealing with larger files. It is important to note that RocksDB limits the file size that can be read to 4GB [9].

3.5 Benchmarking

A set of resource usage monitoring scripts were developed to set up the servers for benchmarking. These scripts were created using the GNU Bash [33] language and provide a comprehensive overview of CPU, memory, and disk usage on the server at 1-second intervals. A sample of the Bash script used for server resource monitoring is shown in Figure 1.

```
#!/bin/bash
# This script monitors CPU and memory usage
# Inspiration from: https://linuxconfig.org/bash-script-to-monitor-cpu-and-memory-usage-on-linux

echo -e "Date, timestamp(s), CPU(%), Memory(MB), TOTAL DISK READ(B/s), TOTAL DISK WRITE(B/s)"
while :
do
    # Get the current usage of CPU and memory
    cpu=$(top -bn1 | awk '/Cpu/ { print $2} ')
    memory=$(free -m | awk '/Mem/{print $3} ')
    dateTime=$(date)
    timestamp=$(date +%s)
    diskWrite=$(iotop -bn1 | awk 'NR==1 { print $12 } ')
    diskRead=$(iotop -bn1 | awk 'NR==1 { print $5 } ')

    echo -e "$dateTime , $timestamp , $cpu , $memory , $diskRead, $diskWrite"

    sleep 1
done
```

Figure 1. Bash script used for server resource monitoring.

The `ab` - ApacheBench benchmarking tool software was utilised for testing. This tool was designed to benchmark HTTP servers. It was initially intended to test the performance of an Apache installation, but it can be used for various use cases. The primary task of this tool is to test how many requests per second an Apache installation or a primary web server can serve [15].

Six different files were created on each server for testing purposes. These files varied in size, with the sizes being 500KB, 1MB, 5MB, 10MB, 20MB, and 50MB. This was done to test the servers' performance when serving different files. As a result, the number of requests made with each of the files varied, with the 500KB file being queried 200 times, the 1MB file 100

times, the 5MB file 50 times, the 10MB file 25 times, the 20MB file 15 times, and the 50MB files ten times.

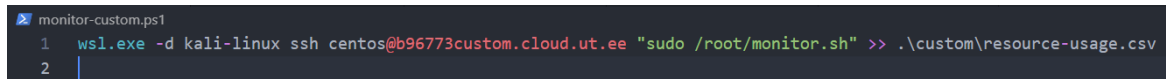
```
benchmark-custom.ps1 > ...
1
2 $nrOfTestsWith500kb = 200;
3 $nrOfTestsWith1mb = 100;
4 $nrOfTestsWith5mb = 50;
5 $nrOfTestsWith10mb = 25;
6 $nrOfTestsWith20mb = 15;
7 $nrOfTestsWith50mb = 10;
8 $url500kb = "b96773custom.cloud.ut.ee/500KB.html"
9 $url1mb = "b96773custom.cloud.ut.ee/1MB.html"
10 $url5mb = "b96773custom.cloud.ut.ee/5MB.html"
11 $url10mb = "b96773custom.cloud.ut.ee/10MB.html"
12 $url20mb = "b96773custom.cloud.ut.ee/20MB.html"
13 $url50mb = "b96773custom.cloud.ut.ee/50MB.html"
14
15 # Test 1
16 Write-Output ""
17 Write-Output "Test GET 500KB.html"
18 Write-Output ""
19 ab -n $nrOfTestsWith500kb -e .\custom\csv-get-500KB.csv $url500kb >> .\custom\get-500KB.txt
20
21 Write-Output "Mean Connection Time (ms)"
22 Get-Content .\custom\get-500KB.txt | grep "Total:" | awk '{print $3}'
23
24 Write-Output "Total Time taken for tests (seconds)"
25 Get-Content .\custom\get-500KB.txt | grep "Time taken for tests:" | awk '{print $5}'
26
27 Write-Output "Failed requests"
28 Get-Content .\custom\get-500KB.txt | grep "Failed requests:" | awk '{print $3}'
29
```

Figure 2. Part of the Windows Powershell script was used to test the Rust server.

The Apache benchmark tool is a command line tool, so the testing process was automated using Windows Powershell scripts [34]. An example of a Powershell script used for testing a Rust server is shown in Figure 2. The script includes the `ab` command for benchmarking, with the `-n` flag specifying the number of requests to be performed and the `-e` flag writing comma-separated values that contain, for each percentage, the time it took to serve that percentage of the requests to the given CSV file. The benchmarking URL is then provided for the program, and the final statistics output is piped to a text file. The full script contains similar test cases for all the file sizes. The entire script is later referred to as a single test suite, and similar test suites were created for testing the Nginx and Varnish installations [15].

An additional Windows Powershell script was created to monitor the resource usage on the remote server. This script utilises an SSH connection to connect to the remote machine and runs the resource monitoring script. The script output is then forwarded to a local file called `resource-usage.csv`, which is later imported into Microsoft Excel to plot the comparison graphs. A sample of the Windows Powershell script used to monitor resource usage is shown in Figure

3.



```
monitor-custom.ps1
1  wsl.exe -d kali-linux ssh centos@b96773custom.cloud.ut.ee "sudo /root/monitor.sh" >> .\custom\resource-usage.csv
2  |
```

Figure 3. Windows Powershell script used to monitor resource usage

The testing was conducted within a home network with a throughput of 200 megabits per second. The system used for benchmarking had an Intel Core i7-9850H processor [35], 32GB of DDR4 RAM, an Intel Wireless-AC 9560 network adapter [36], and a Windows 10 Pro 64-bit (10.0, Build 19045) operating system. The ApacheBench software used for testing was version 2.3 Revision 1901567.

To benchmark a single server, the resource monitoring script was executed, followed by the benchmarking Powershell script to begin the benchmarking process. Once the testing was complete, the results from the resource monitoring script and the benchmarking scripts were collected into Microsoft Excel.

4 The Results

The study involved running test suites three times on each caching solution and taking the average of the results. The test suite was considered successful only when there were no failed requests, as failed requests can substantially affect the time it takes to serve all requests. The resulting graphs and tables present the averaged results across all test suites, with the keyword "Custom" referring to the RocksDB and Rust solution.

File size	Average across all test suites					
	Mean Connection Time (ms)					
	500KB	1MB	5MB	10MB	20MB	50MB
Custom	264.67	321.33	646.33	1055	1411	3257
Nginx	283.33	374.67	972	1712	2680.67	6045
Varnish	281.67	376	996	1621.33	2968.33	5577.33
File size	Total Time Taken For Tests (seconds)					
	500KB	1MB	5MB	10MB	20MB	50MB
Custom	52.912	32.161	32.323	26.381	21.165	32.569
Nginx	56.604	37.486	48.598	42.799	40.21	60.451
Varnish	56.322	37.591	49.788	40.532	44.526	55.772

Table 1. Averaged results across all test suites

Table 1 shows that the custom caching solution outperformed Varnish and Nginx in all test cases. Notably, as file sizes increased, the speed difference between the custom solution and the others increased even further. In particular, when querying files with a size of 50MB, the custom solution was nearly twice as fast as Nginx.

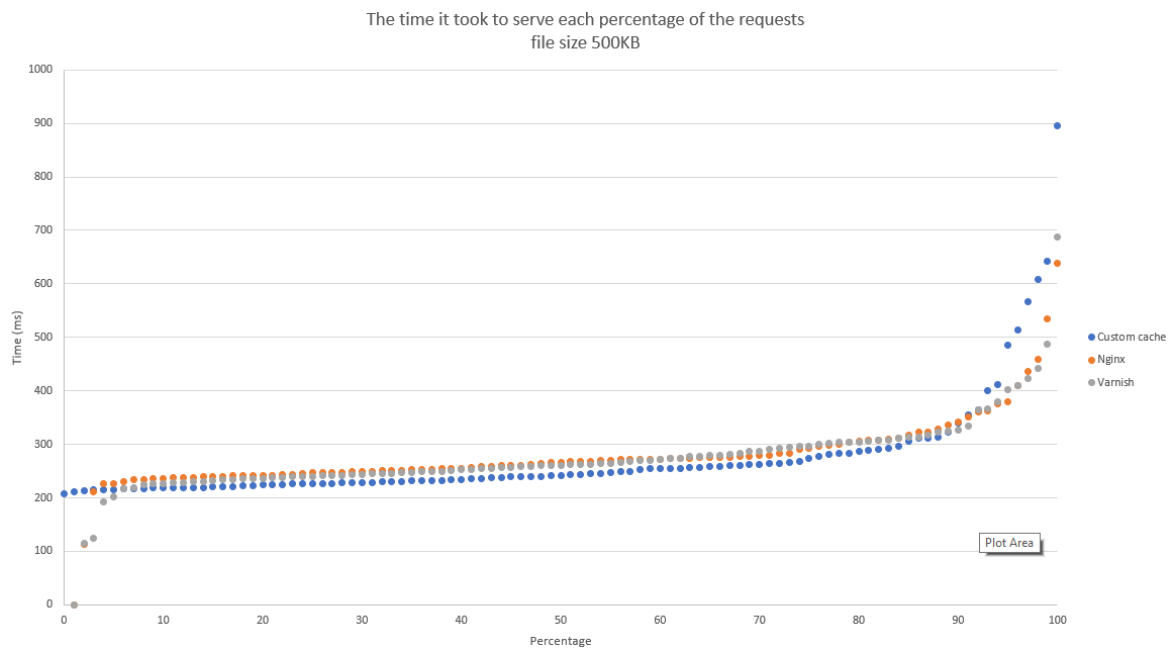


Figure 4. 500KB file size speed comparison under load

Figure 4 illustrates that the performance of the three caching solutions is similar when used to serve small files. However, when serving larger files, the custom caching solution outperformed Varnish and Nginx almost throughout the test case, with the speeds of the caching solutions decreasing towards the end. This performance decrease hints at a possible decrease over time under continuous load, an exponential pattern in all caching solutions.

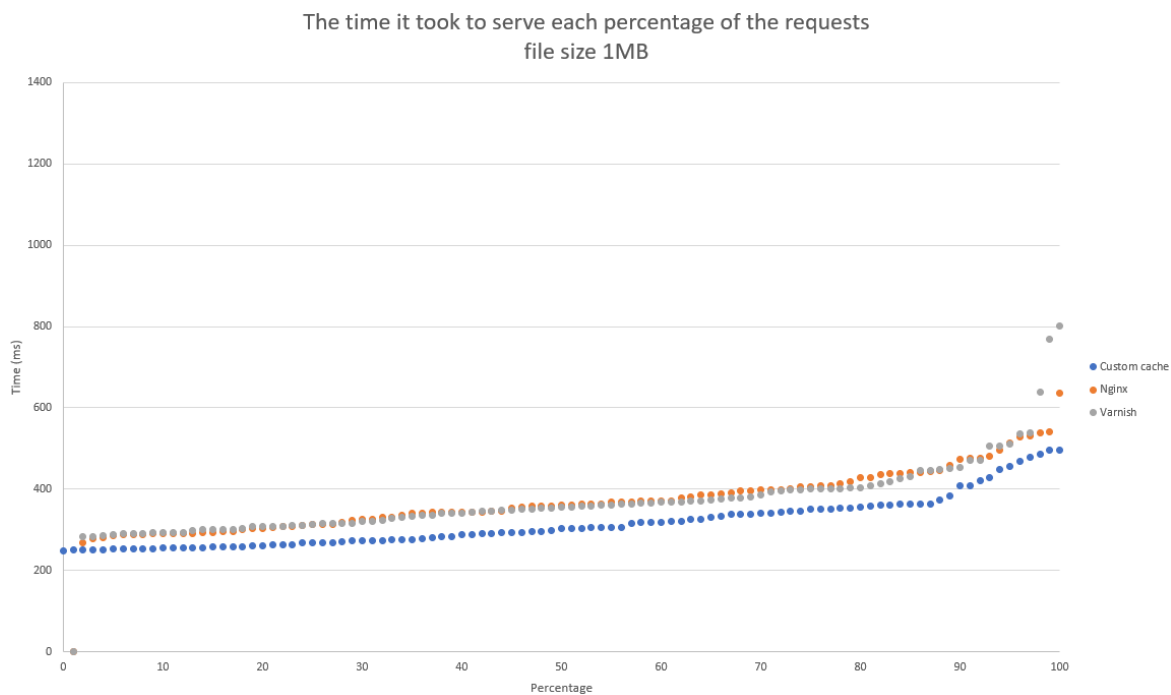


Figure 5. 1MB file size speed comparison under load

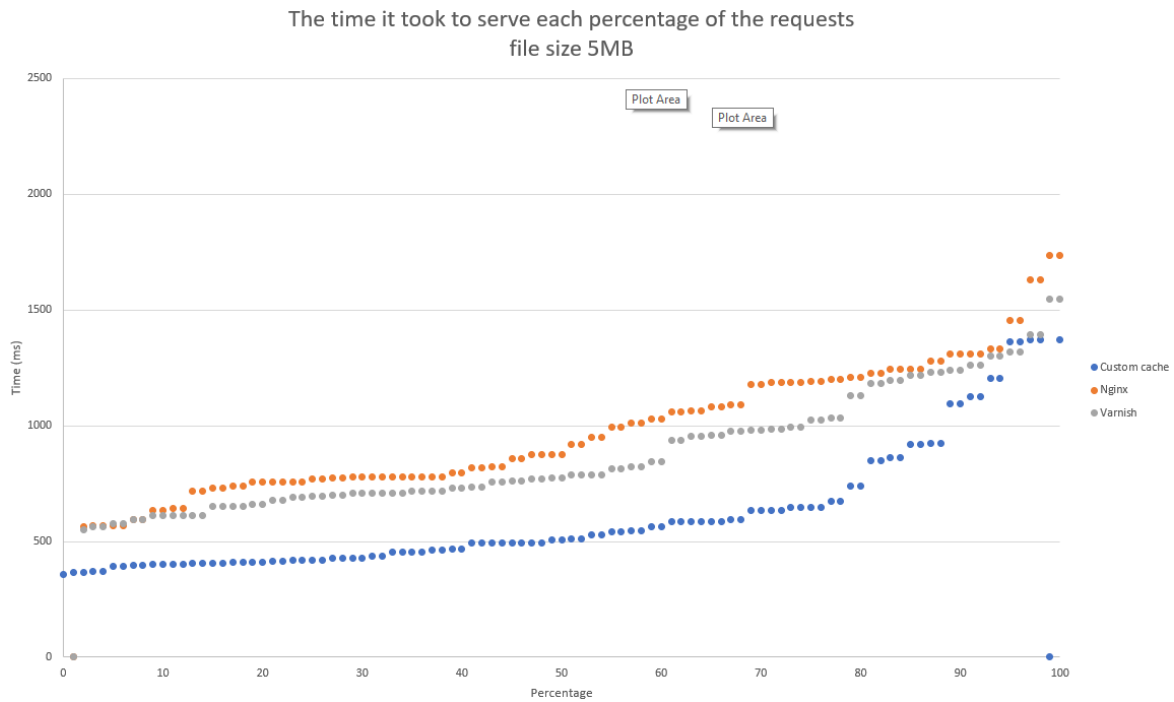


Figure 6. 5MB file size speed comparison under load

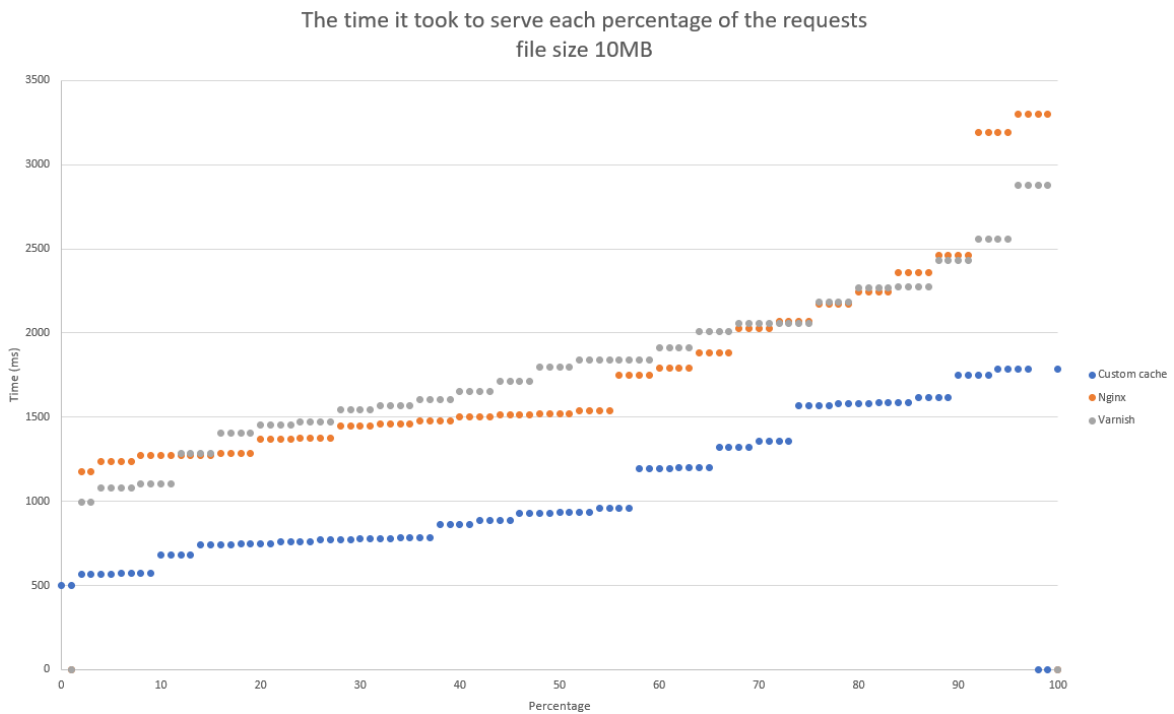


Figure 7. 10MB file size speed comparison under load

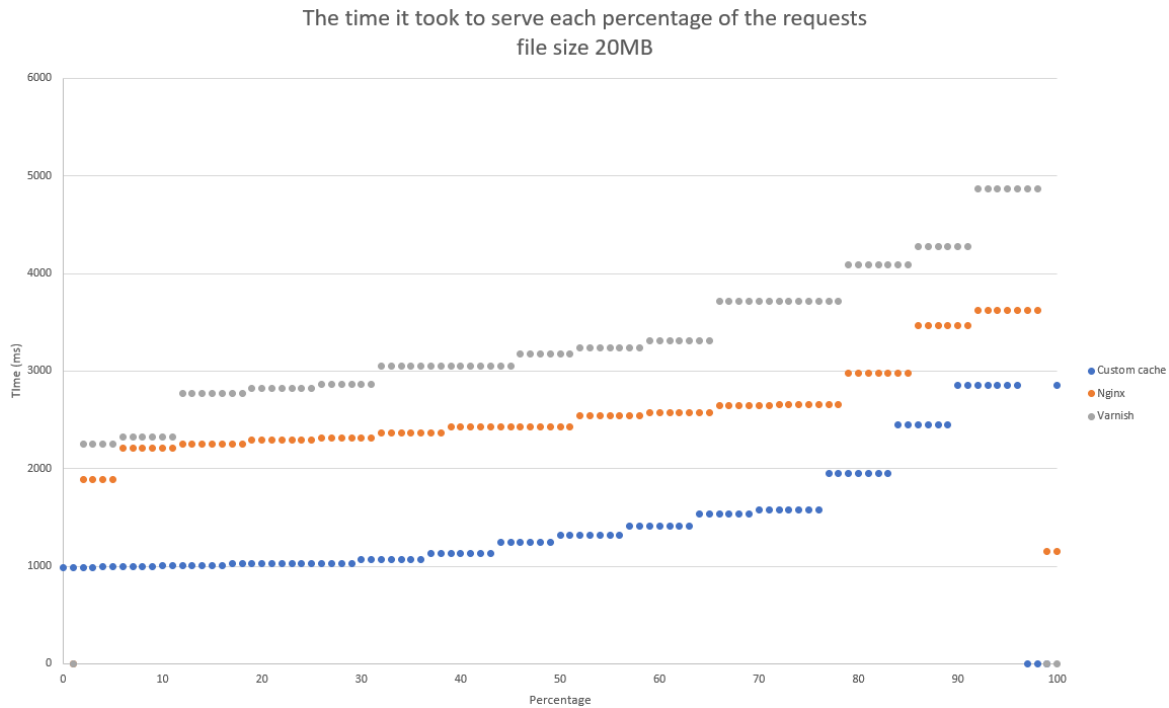


Figure 8. 20MB file size speed comparison under load

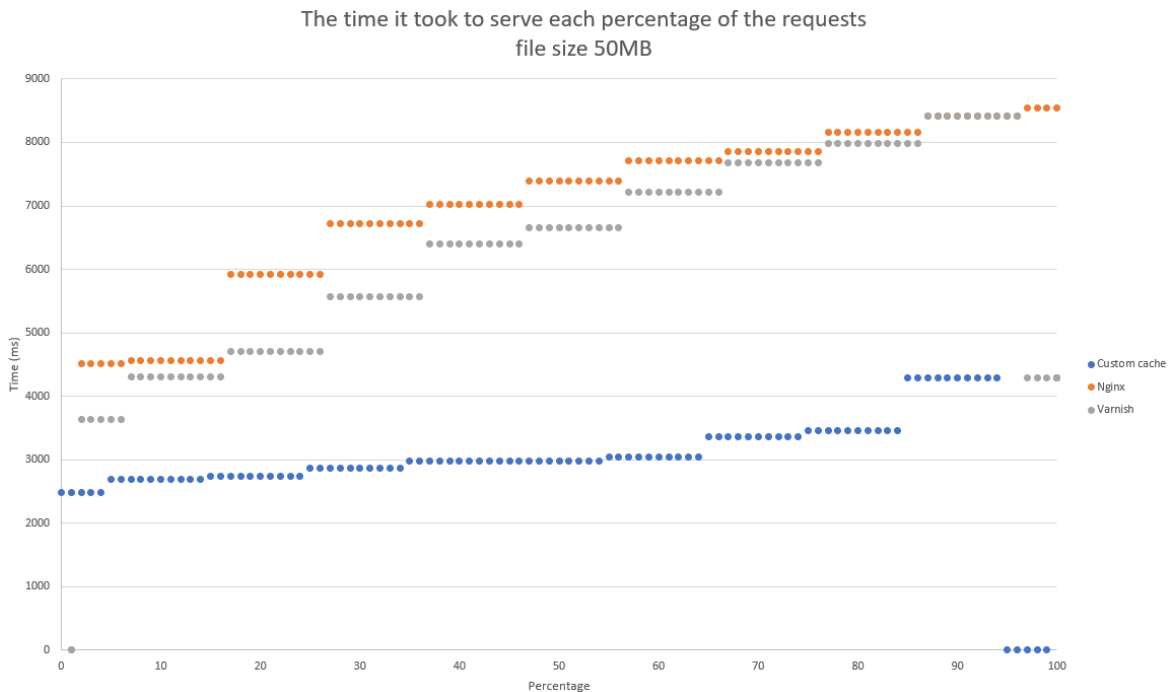


Figure 9. 50MB file size speed comparison under load

Figures 5 through 9 further compare speed under load for different file sizes, with the custom caching solution consistently outperforming the other solutions, especially for larger file sizes.

However, there are some drawbacks to the custom solution, as shown in Figures 10 and 11.

First, the custom solution uses nearly twice as much memory as Nginx. Moreover, Varnish and the custom solutions store the entire file in RAM before sending it via HTTP. This can become problematic when caching larger files that may not fit in memory, such as larger multimedia files. Nginx does not have this problem, as its memory usage was stable during the entire test suite. Additionally, the custom cache implementation experiences seemingly random CPU usage spikes during the test suite, while Nginx has stable performance in both RAM and CPU usage.

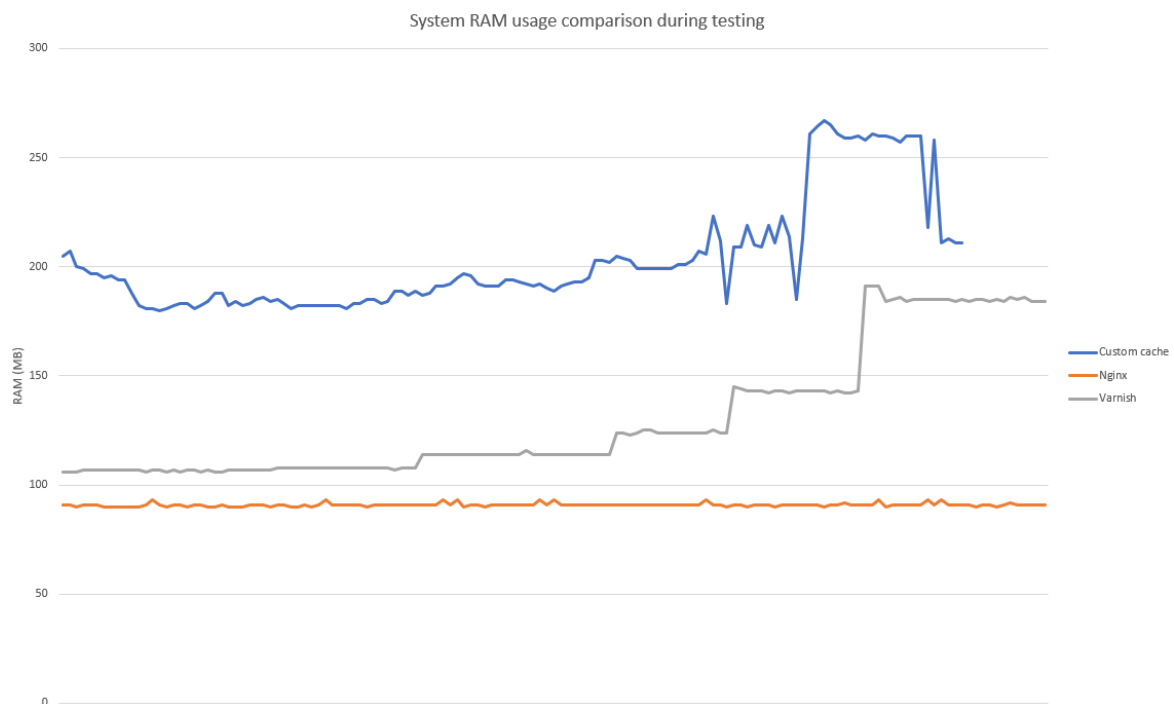


Figure 10. System RAM usage comparison during testing

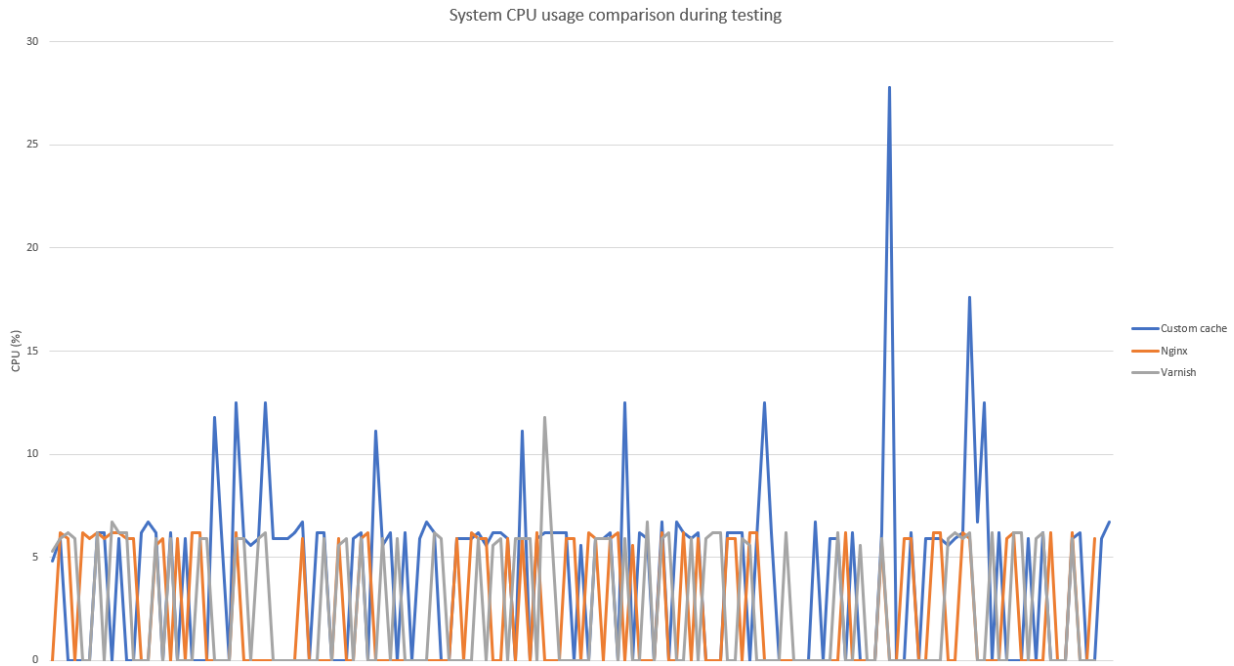


Figure 11. System CPU usage comparison during testing

Based on the previous results, the custom caching solution using RocksDB and Rust outperformed both Varnish and Nginx regarding speed for serving large files but had drawbacks regarding memory usage and CPU spikes. Therefore, there are several practical implications of these findings for CDN developers.

First, CDN developers who prioritise fast serving of large files should consider using the custom caching solution presented in the study. However, they should also know this solution's potential memory and CPU usage issues and ensure their hardware and software configurations can accommodate them.

Second, CDN developers considering implementing a caching solution should consider the performance characteristics of different solutions and their specific needs. For instance, if their server predominantly serves small files, they may find that Varnish or Nginx provides adequate performance without the custom solution's added memory and CPU overhead. However, suppose their CDN serves large multimedia files. In that case, they may need to consider implementing the custom caching solution introduced in this work or optimising their server infrastructure to handle the demands of large file caching.

Third, the cost and feasibility of implementing the custom caching solution should also be considered. While the solution may provide superior performance, it may require additional

hardware or software resources to accommodate its memory and CPU usage needs. Therefore, CDN developers should weigh the potential benefits of increased speed against implementation and maintenance costs.

Overall, the practical implications of the findings suggest that CDN developers should carefully consider their specific needs and the performance characteristics of different caching solutions when choosing a caching solution for their server. They should also be aware of the potential drawbacks associated with each solution and ensure that their infrastructure can accommodate the demands of the chosen solution.

5 Conclusion

This thesis aimed to develop and benchmark a custom caching solution that can compete with commercial products like Nginx and Varnish. The custom caching solution was implemented using Rust programming language with a RocksDB embedded database for fast content caching. The results of the tests and analysis have shown that the custom caching solution outperforms varnish and Nginx in raw speed in all the test cases.

The findings of this thesis suggest that the developed custom caching solution is a viable alternative to commercial caching solutions. The test results analysis reveals that as the file sizes get more significant, the speed difference between the custom solution and the others increases even further. For example, when querying files with a size of 50MB, the custom solution is almost twice as fast as Nginx. This does come with a drawback of increased RAM usage and seemingly random CPU usage spikes under load. None were present on Nginx, which provided slower speeds but stable performance under load.

These results have important implications for the development of future CDN solutions. The custom caching solution developed in the thesis can be a foundation for building new and more efficient CDN solutions. Moreover, this study's findings can help select the best caching solution for websites with large file sizes.

In conclusion, this thesis has successfully developed and benchmarked a custom caching solution that outperforms commercial products like Nginx and Varnish regarding speed. The developed solution has the potential to provide significant improvements in the performance of web servers. It can be a foundation for building new and more efficient CDN solutions. Future research can build on these results and further improve the performance of web caching solutions.

Acknowledgements

I am grateful to my supervisors, Jefim Gasels and Ahti Saar, for their guidance, support, and expertise throughout my thesis's practical and theoretical parts. Their patience and feedback have helped me navigate the challenges of this study.

A handwritten signature in black ink, appearing to be 'CL' or similar, located at the bottom left of the page.

Bibliography

- [1] Gang Peng, "CDN: Content distribution network," *arXiv preprint cs/0411069*, 2004.
- [2] Jeff Chase, Michael Rabinovich Syam Gadde, "Web caching and content distribution: A view from the interior," *Computer Communications*, vol. 24, no. 2, pp. 222–231, 2001.
- [3] Thomas and Jain, Shruti and Andra, Usha, and Khurana, Taru Barnett, "Cisco visual networking index (vni) complete forecast update, 2017--2022," *Americas/EMEAR Cisco Knowledge Network (CKN) Presentation*, pp. 1--30, 2018.
- [4] Abhigyan and Venkataramani, Arun and Sitaraman, Ramesh K Sharma, "Distributing content simplifies isp traffic engineering," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, pp. 229--242, 2013.
- [5] F5 inc. Igor Sysoev Netcraft. Nginx Plus Documentation. [Online]. <https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/#purging-content-from-the-cache>
- [6] Varnish Software AB. Varnish HTTP cache. [Online]. <https://varnish-cache.org/>
- [7] Rust language. [Online]. <https://www.rust-lang.org/>
- [8] Facebook. RocksDB. [Online]. <http://rocksdb.org/>
- [9] Facebook. Github, RocksDB wiki. [Online]. <https://github.com/facebook/rocksdb/wiki/>
- [10] Steve and Nichols, Carol Klabnik, *The Rust Programming Language (Covers Rust 2018)*:. No Starch Press, 2019.
- [11] megajakob. (2020, November) How to build your own CDN. Web. [Online]. <https://dev.to/megajakob/how-to-build-your-own-cdn-io1>
- [12] Sissel-Johanne Aspdal, Optimized use of CDN technologies in Uninett, 2020.

- [13] OpenAI. OpenAI, ChatGPT. [Online]. <https://openai.com/>
- [14] Grammarly Inc. (2023) Grammarly, Communication Assistance. [Online]. <https://www.grammarly.com/how-grammarly-works>
- [15] The Apache Software Foundation. Apache HTTP server project, ab. [Online]. <https://httpd.apache.org/docs/2.4/programs/ab.html>
- [16] Estonian roadmap of research infrastructures. Estonian Scientific Computing Infrastructure. [Online]. <https://etais.ee/>
- [17] Red Hat Software. Red Hat Ansible. [Online]. <https://www.ansible.com/overview/it-automation>
- [18] Red Hat Software. Extra Packages for Enterprise Linux (EPEL). [Online]. <https://docs.fedoraproject.org/en-US/epel/>
- [19] The Apache Software Foundation. Apache HTTP server project. [Online]. <https://httpd.apache.org/>
- [20] Netcraft, F5 Inc. Igor Sysoev. Nginx Documentation, ngx_http_proxy_module. [Online]. https://nginx.org/en/docs/http/ngx_http_proxy_module.html
- [21] Netcraft, F5 Inc. Igor Sysoev. Nginx Documentation, Nginx Content Caching. [Online]. <https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/>
- [22] James Henstridge. Python pygpgme module. [Online]. <https://pypi.org/project/pygpgme/>
- [23] Varnish Software AB. Varnish Developer Portal, Installing Varnish on Centos. [Online]. <https://www.varnish-software.com/developers/tutorials/installing-varnish-centos/#4-configure-varnish>
- [24] Rust Team. Getting started with Rust. [Online]. <https://www.rust-lang.org/learn/get-started>
- [25] Facebook. Installing RocksDB. [Online]. <https://github.com/facebook/rocksdb/blob/main/INSTALL.md>

- [26] Christoph Galuschka. CentOS Product Specifications. [Online]. <https://wiki.centos.org/About/Product?highlight=%28compiler%29>
- [27] Trevor Hemsley. CentOS wiki, The Software Collections (SCL) Repository. [Online]. <https://wiki.centos.org/AdditionalResources/Repositories/SCL>
- [28] Trevor Hemsley. CentOS Wiki. [Online]. <https://wiki.centos.org/HowTos/SELinux#Introduction>
- [29] systemd System and Service manager. [Online]. <https://www.freedesktop.org/wiki/Software/systemd/>
- [30] Sean McArthur. Github, hyper Rust library. [Online]. <https://github.com/hyperium/hyper>
- [31] Tokio Contributors. Github, tokio. [Online]. <https://github.com/tokio-rs/tokio/blob/master/LICENSE>
- [32] Carl Lerche. Github, async-stream. [Online]. <https://github.com/tokio-rs/async-stream/tree/master>
- [33] Free Software Foundation. GNU Operating System, GNU Bash. [Online]. <https://www.gnu.org/software/bash/>
- [34] Microsoft Corporation. (2006, Nov.) Windows Powershell Documentation. [Online]. <https://learn.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.3>
- [35] Intel Corporation. Intel® Core™ i7-9850H Processor specification. [Online]. <https://ark.intel.com/content/www/us/en/ark/products/191047/intel-core-i79850h-processor-12m-cache-up-to-4-60-ghz.html>
- [36] Intel Corporation. Intel® Wireless-AC 9560 Specification. [Online]. <https://www.intel.com/content/www/us/en/products/sku/99446/intel-wirelessac-9560/specifications.html>

Appendices

1 GitHub repository links for the projects

1. Rust caching server repository link:

<https://github.com/Chapslock/thesis-cache-server>

2. Ansible project used for automation:

<https://github.com/Chapslock/thesis-ansible-project>

3. Windows Powershell tools used for benchmarking:

<https://github.com/Chapslock/thesis-benchmarking-tools>

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Charl Kivioja,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

“Efficient caching of web content”

supervised by Jefim Gasels and Ahti Saar

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe on other persons' intellectual property rights or rights arising from the personal data protection legislation.

Charl Kivioja
10.05.2023