Final Year Project

A Framework For User Specified Linux Testing

James Kirwan

Student ID: 17402782

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Professor Lastovetsky

UCD School of Computer Science

University College Dublin

March 9, 2022

Table of Contents

Abstract

In recent years, Linux operating systems have seen increased usage across the world. These operating systems vary in performance from distribution to distribution. A major advantage of Linux operating systems is that they are open source, and allow anyone to contribute to their development. However, while there does exist a number of benchmarking suites for these systems, there does not exist a convenient way to benchmark and compare performance between builds and distributions.

I am proposing an automated Linux framework, that allows a user to specify which aspects of particular systems they wish to test. The system would then take these systems and run the user-specified tests on them. Upon completion of these tests, the system would display the desired results, with comparisons between builds. The user will be able to specify what data to view from a front-end web page.

In short, this project aims to use system scripting and benchmarking programs and to develop algorithms to analyse their results. The end goal would be a fully automated system that allows for user input when testing the system will produce data suggesting the optimal configurations of all systems tested.

Chapter 1: Project Specification

According to Distrowatch.com[1], there are hundreds of Linux distribution in use today. When a user wants to choose an optimal configuration of Linux that suits them, the amount of options can be confusing. To help make this decision, benchmarking software has been created to test different aspects of a system. However, these benchmarks only produce raw data. It is up to the user to interpret this data and make their own observations. The amount of data produced this way and how it is presented can make choosing their configuration more confusing. In short, the problem this project aims to solve is the complexity of testing multiple configurations of Linux and finding an optimal configuration.

The goal of this project is to simplify the process of finding the optimal configuration of Linux for a user. To achieve this goal the following objectives will be reached:

Core Requirements

- A method for specifying and installing tests will be developed.
- A means of configuring and then running tests will be created. The data from these tests will be collected.
- The optimal configuration(s) from the tests run will be identified for the user.

Advanced Requirements

- A graphical display of the optimal configurations will be created.

Chapter 2: **Introduction**

In recent years there has been an increase in the use of Linux operating systems across a range of industries.[2] These systems are open source, allowing anyone to read the source code and even to contribute them. As these systems, or distributions as they are also known as, release new builds, performance can change for better or worse. To help measure performance and gain metrics of these systems, benchmarking suites have been created to test different attributes and provide measurable results. By examining the performance of different distributions, it may be possible to gather useful information for a Linux user when deciding what distribution to use for different purposes.

More recent benchmarking suites, such as Phoronix[3] or Sysbench[4] allow for more detailed testing of such systems. Features like automated testing or using graphical user interfaces to display test results are employed to make the process of testing a Linux system more convenient. The problem for a user that is testing multiple configurations of Linux is that even with the raw data presented in front of them, the most optimal configuration that they tested may not be immediately evident.

The goal of this project is to examine the performance of Linux distributions in a user-specified manner. This would mean designing a framework where a user can specify what systems to examine and what aspects to test. Additionally, the framework should analyse these test's results and perform an analysis to identify optimal configurations under the user's parameters. The project will require advanced usage of scripting languages such as Bash, as well as a keen interest in Linux operating systems and their functionality. Additionally, data analysis techniques such as multi-objective optimization will be required to detect the best configurations of these systems.

In the following sections the related works will be examined, the approach will be outlined, the implementation will be discussed in detail and evaluated. Lastly, any conclusions and future works will be considered.

Chapter 3: Related Work and Ideas

In this chapter, we examine similar works that exist on the topics of benchmarking systems, analysing performance, and displaying these findings to a user. The main point of focus will be on how a user can input their own preferences into the testing process, and in what results they wish to be displayed at the end of the process.

3.1 Benchmarking

In this section, popular Linux benchmarks will be examined. When looking at which benchmarks to compare this project against, the criteria will be that the software must be open source, and they must be capable of being run through, and output to, the command line interface. Testing of these benchmarks was done on a system with an eight core i5 processor, with 8 gigabytes of RAM and a 256 gigabyte SSD.

3.1.1 UnixBench

UnixBench is the longest running Unix benchmarking suite.[5] Originally designed in 1983 by Monash University, it has since been developed by BYTE magazine, and is also contributed to as an open-source project. The goal of UnixBench is to benchmark system performance, not the performance of any particular piece of hardware. This means that the results of the test will be dependent on the operating system, compiler and libraries as much as they are on hardware.

The types of test that comprise UnixBench test the performance of system processes. Some attributes that are tested are the likes of file copy speed, pipe throughput, process creation, floating point operations, shell script performance and execl throughput. Some hardware is tested, Dhrystone is executed as the first test to check CPU performance and a series of graphical tests are run to check the performance of 2D and 3D graphics.

All tests are with access to a single core first, and then they are each run with a copy of the current test on each core. The results of these tests are compared against a baseline system. Upon finishing all tests, the results are outputted to the command line interface, and to a plain text log file, and a html file.

In terms of user specification, UnixBench does not offer much. A user can run the suite in quiet or verbose modes depending on how much detail they need at execution time. They can also specify the number of iterations per test, and the number of copies of each test to run in parallel. The user cannot choose to exclude particular tests or change run times for particular tests without editing the project's code.

In regard to this project, UnixBench could be included as a modular component for testing system resources. However, the lack of options a user has for running tests make it less suitable to the project goal. When testing Unixbench, the whole suite took 55 minutes to run on average with default settings.

3.1.2 Sysbench

Sysbench is an open-source, multi-threaded cross-platform (Windows, Linux, macOS) benchmark[4] suite developed by Alexey Kopytov.The suite contains tools designed for primarily testing databases under intensive load, but they can also be used for more general system testing. The features currently installed in Sysbench aim to test file I/O, scheduler, POSIX threads and database server performance as well as memory allocations and transfer speed.

Execution of these tests is done through the command line interface, where the Sysbench command is used followed by any qualifiers the user wants for the test execution. Some qualifiers are specific to certain tests, and others are more general, for example the number of threads assigned to a test. A feature present in Sysbench that was absent in UnixBench was an ability to define a max time for the tests to run in. Sysbench also includes a batch mode. Batch mode allows a user to specify an amount of time in seconds, and for every repetition of that amount of time, the minimum, maximum and percentile values for the current test will be printed to the output.

Sysbench's tests are not run in an automated fashion, and it is the most manual of the benchmarking suites observed in this section. Each test must be specified one by one, and some tests require the use of the 'prepare' command which will create test data. An example would be the fileio test, which requires a test set of 128 files, totalling 150 gigabytes, to be created before running. Upon completion of these tests, it is recommended that the cleanup command is run to remove any temporary test data.

Running each of the different tests within Sysbench revealed that execution times vary broadly on a test-by-test basis, with the likes of the CPU test running in seconds, but the fileio test taking twenty minutes with preparation and cleaning time included.

Examining Sysbench through the lens of this project reveals features to improve upon and to implement. Under the goals of this project, the manual elements of Sysbench should be ignored or automated, such as the preparation and clean up procedures. Allowing the user to specify the amount of time allotted to the benchmark is an interesting idea and would benefit the project's goal of user specified benchmarking. Lastly, on the idea of including Sysbench as a module in the project, it does not look fitting. A lot of shell scripting and engineering would be needed to make the suite fit the vision of the project.

3.1.3 Phoronix Test Suite

The Phoronix Test Suite is an open source, cross platform automated bench marking suite that gives a user access to a wide array of individual benchmarks and whole suites that be installed in a modular fashion.[3] Phoronix is designed to be easy to use, it handles the installation and dependency management of each test and records test results.

Running of tests can be handled in three ways, single tests, whole test suites, or batch jobs. Individual tests and test suites can be viewed from the command line interface once Phoronix has been installed. All tests are run from the CLI. Running a singular test is done by using the run test command followed by the name of the test. The same is true for running a whole suite. Where batch jobs differ is they allow the user to specify which parts of the Phoronix test process that they want to run. For example, disabling prompts during the running of tests, enabling / disabling test options, or uploading results. This customisation of the test environment is of particular interest for this project.

The record keeping of Phoronix also contains several options that can be specified by the user. The test suite can archive system and installation logs, and most importantly, the test results. When

logging the results, the user has the option of keeping the logs in plain text format, as a PDF or in a web-based results viewer. The results can also be uploaded to Openbenchmarks.org[6], a website that provides test storage as well as collaboration tools for comparing results. The website can also be used for analysing results of tests, and sharing the performance of hardware on Linux systems, as well as sourcing prices.

The feature set of Phoronix makes it a good point of comparison for this project. The breadth of different tests it offers also make it a good choice for inclusion, as specific tests can be run to complement other test suites previously mentioned. Features like the batch jobs would allow a user greater choice when specifying how they would like to run.

3.2 Performance analyses

Benchmarking systems has no merit unless some goal or observation can be reached. In this section, we will examine how other similar works analyse their results and what kinds of tools are used for displaying the end product.

3.2.1 Comparing Performance Between Distributions

This year, the Faculty of Electrical Engineering in the University of Osijek produced a paper comparing performance between three popular Linux distributions from 2020[7]. They used a series of benchmarks and outlined a methodology for testing the performance of Ubuntu 20.04 LTS, Linux Mint 19.3 and Pop!_OS 20.4. Their approach and methodologies are of particular interest for this project.

The methodology of their testing was as follows. Firstly, three benchmarking tools were selected, Hardinfo, Geekbench and Phoronix. Hardinfo and Geekbench were used to test CPU performance and Phoronix was used to evaluate SSD, RAM and GPU performance. The performance of each system was measured on a high-performance computer, with each version of the three distributions being 64-bit. Before running their benchmarks, they developed a performance measurement methodology to ensure reliability and consistency. A flowchart was developed to outline the testing process, with a formula to evaluate the performance of their model as it ran through five repetitions. Lastly, they chose one system, Ubuntu in this case, as a baseline to compare the results of the other distributions against.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \rightarrow N = 5$$

$$x_{imin} \leq \bar{x} \leq x_{imax}$$

Figure 3.1: Performance measurement Methodology Formula

$$Diff_{\%} = \frac{\text{Mint\_}or\text{\_Pop\_}val - \text{Ubuntu\_}val}{\text{Ubuntu\_}value} x100\%$$

Figure 3.2: Performance Comparison Formula

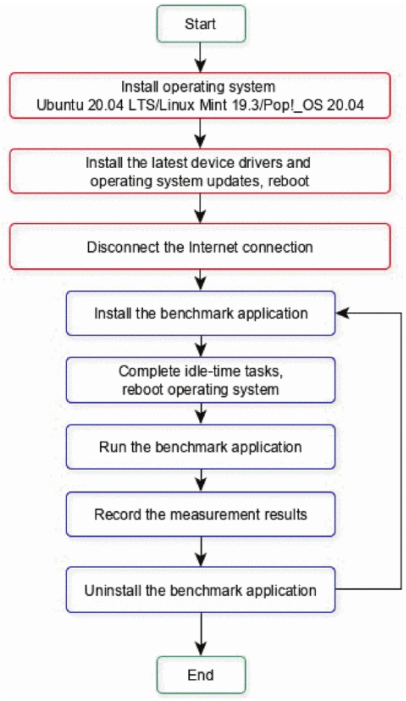

Figure 3.3: Testing Procedure Flowchart.

This testing procedure was used for each distribution, with only the last 5 steps (in blue) being measured. The overall performance was evaluated by three attributes: Speed, Time and Points. Speed was the number of achieved units in a period of time, measured in Mb/s. Time was the amount of time taken to complete a task, and Points were determined by the formula in Fig 2.2.

This paper is of great relevance to this project as it outlines a particular methodology that can be deviated from to consider automation. A similar point of comparison would be the goals, whereas this experiment was concerned with finding the performance differences of just three distributions, this project is concerned with both that goal and finding performance differences between builds of a particular distribution. In short, development of a performance measurement methodology, a of points system to measure success, and a clear flowchart of procedures should be considered when implementing the end product.

3.2.2 Multi-Objective Optimization

Multi-objective optimization is a field of Mathematics concerned with modelling the behaviours of complex systems to discover optimal methods for performance. An article in Elsevier's Computers and Mathematics with Applications journal[8] compares a number of multi-objective optimization methodologies. The researchers identified two problems in optimizations, single-objective and multi-objective optimizations. It is important to first understand the goal of single-objective optimization before expanding to multiple objectives.

To achieve a single-objective optimization is to find the best solution for one metric. In the scope of this project, that would be a single element of a Linux system, such as CPU usage or file I/O. In their article, the researchers define a general function for an optimization plan. In this plan, a scalar objective function $f(x)$ is created and is minimized or maximized depending on a series of inequality constraints G_m, where m is the total number of inequality constraints. Inversely to G_m, H_p are the equality constraints with the total number of equality constraints being p The value

x is a vector that contains all relevant results that can be evaluated for this optimization. The variables in x can be continuous or discrete, and the total number of variables in x are given in n. An important limit when defining constraints is that $p < n$ must be true, or else the problem is defined as overly constrained. The constraints are designed to give an end result in line with the overall goal.

Multi-objective optimization expands on this concept by creating a vector function $F(x)$ that consists of $f_k(x)$ functions. An issue that separates the multi-objective optimization from the single-objective is that the number of solutions considered in the former are potentially uncountable. $F(x)$ must consider each component vector from $f_1(x), ..., f_k(x)$ and map each to an axis, while each individual vector must consider their own amount of decision variables in x and map them to their own respective axis. Due to the number of objectives at play, it is recommended that Pareto-Optimality theory is applied to obtain desired results. This theory reasons that to achieve the desired result for one attribute, another attribute will have to be reduced.[9] The final component of the multi-objective optimization problem is a decision maker. This component selects the Pareto optimal solution, the solution that is the best compromise of all desired objectives.

The paper goes on to suggest a number of multi-objective optimization methodologies that may be suitable for this project. The key takeaway point from this paper is that it gives an idea on how results can be analysed, and optimal solutions may be discovered. When manipulating the results of benchmarks, they should be presented in a manner that is suited to this form of optimization, and a user should be allowed to specify a preference for Pareto-optimality.

3.2.3 Displaying Results

A number of open-source analytics tools exist on the Internet. Grafana is one such tool that is used by thousands of businesses and users from across the world.[10]

Grafana is an analytics platform that allows a user to query and visualise their metrics in a highly customizable dashboard[11]. Each dashboard consists several panels that each has a query to a specific data source. The query data can be then used to produce different types of graphs, such as histograms, line graphs or scatter plots. The data sources can be obtained from a number of configured Data Sources such as Graphite or MySQL.

The process in which Grafana takes information from a data source is all done inside of the panel creation. Firstly, a user chooses a data source to query from. They then write their query. The next step is visualisation, a wide range of options will be shown that are pertinent to the particular data that the query is returning. Once the user is satisfied with their visual representation of their metric, they can save the panel and place it on the dashboard.

To conclude this examination of Grafana, the process in which results are queried and displayed are of particular interest to this project. The need for a database to store test results and query them into some form of graph will need to be considered when developing the front end of this application.

3.3 Summary of Related Works

In these related works we have seen that there are a wide variety of benchmarking suites freely available in today's market. These benchmarks allow for testing of specified system attributes, or in the case of Phoronix, modular installation of tests or whole suites with automation options if the user wishes for it. In the Performance analyses section, a methodology for testing Linux systems was examined and can be further built upon as needed for the goals of this project. Furthermore, by examining the subject of multi-objective optimization, a clear means of suggesting the best performance between systems, and allowing the user to specify their desires, can be developed. Using this research, this project will develop a fully automated testing suite that will discover performance differences between concurrent builds of Linux systems or different distributions entirely and then present the results in a readable manner.

Chapter 4: Outline of Approach

To begin the project, the first element considered was the benchmark that would be used. Benchmarking was chosen as a starting point because a set of results was needed later in the project to choose an optimal configuration. When choosing a benchmark to use, the options previously examined in chapter 3 were considered. Ideally, the framework should work with any benchmark a user wishes to use, but to achieve the goals of this project in the time frame given, one benchmark was chosen. The software chosen was Phoronix.

This benchmarking software was chosen out of the others is due to the breadth of tests it can run. User specified testing is a goal for this project, so using the software with the most choice of tests that can be run was deemed the most acceptable. Furthermore, when considering how the project will be evaluated, a larger variety in tests would allow for better analysis of the multi-objective optimization algorithm used to select the best configuration.

With the benchmark implemented in the framework, additional smaller features were considered to give the users more input into the testing process. When considering features to add, the goal was to make the framework more convenient to use and not to increase the complexity of use. As such, the additional features that were added allowed the user to access certain parts of the framework, like installing new tests or running pre-made test suites, rather than having to run the whole framework over again. The choice to add these features became apparent when developing the framework, as it was more time efficient to access only certain features for testing during development than to run all processes from the start. It occurred to me that the user may also want this functionality, especially if they want to run repeated tests.

After these features were implemented, the next step was to create a way to select an optimal configuration from the tests that were run. The methodology in section 3.2.2 was consulted to get an idea of how to best approach the problem of choosing an optimal configuration. Applying the theory from that paper proved difficult at first, so it was suggested to start with a limited number of objectives to optimise. By doing this, I was able to understand how the theory can be applied in the context of this project and then gradually expand the amount of objectives that could be considered.

At this point in the project, it became important to clearly define what an objective is in relation to the framework. For the purposes of this project, an objective is a metric specified by the user in which they want to achieve a desirable state or performance in. The metrics in the scope of this project are the output of the tests run.

From gradually increasing the objectives, it became apparent that each test run was an objective to be optimised. This indicates that if a user was to run three tests, a tri-objective optimization would need to take place. To find the optimal configuration of these objectives, we would want to find the Pareto optimal[12] set of objectives. This implies that the most optimal configuration of all the configurations tested is the set where each objective achieves the greatest performance without impacting the performance of other objectives.

In its finished state, the project can be used for N-objectives, but as N increases the program takes longer to run.

The last stage of the approach was to create a way to visualise the results. The way the results were conveyed to the user changed with each iteration of the project. Initially, when only two objectives were being considered, a simple graph was created. The graph would show all of the

results from the tests that were run, with each objective making an axis of the graph. All test results would be displayed on the graph, with the optimal results being highlighted in a different colour.

The tri-objective representation was similar, but with an extra axis. For objectives past the three, only a file containing the optimal configurations was created, as displaying data beyond the third objective would take more time than the project had left. Additionally, the file containing optimal configurations was created for bi- and tri-objective configurations, as the graphs may be difficult for someone with colour blindness to read.

Below (Fig 4.1) is an overview of the scripts, programs and directories made for the project

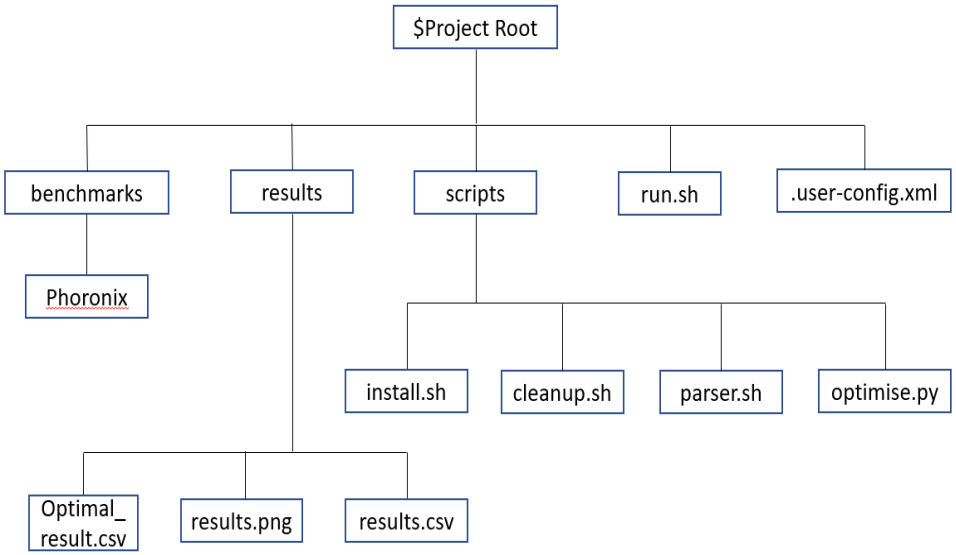

Figure 4.1: High level Diagram of project files

Chapter 5: Detailed Design Implementation

In this chapter, the implementation of the framework will be discussed in detail. Important sections of code will be highlighted and explained. How the features were developed and any problems encountered during the realization of the project will also be discussed.

This project was developed using Bash scripting and Python. Bash was used for file manipulation and executing benchmarks. Python was used to implement the multi-objective optimization program.

5.1 Installation of Benchmark Suite

The first step taken to implement the project was the creation of an installation script. The goal of this script was to install the benchmarking suite chosen, and to facilitate the installation of any tests a user may want to run. Benchmarks are stored in a directory named Benchmarks in the project's root folder. The installation of the benchmark is done by accessing the software's own installation process. Before doing this, the framework ensures that the user has root privileges so that the benchmark may be installed properly. Upon installing the benchmarking tool, a hidden file named ".token" is created so that this step may skipped on repeat runs of the framework.

As mentioned in the approach, this project was developed to work with one benchmarking suite to achieve the goals set out in the given time frame. Phoronix, the chosen software, allows for the installation and running of over one hundred tests[13]. A method for allowing the user to specify which tests they wanted to run had to be developed. The approach taken was to create a test file(Fig 5.1). This test file took the form of a plain text file that instructs the user to specify which tests they want Phoronix to install on a separate line.

```
#Enter the tests you wish to install on their own line

pts/scimark2
pts/fio
pts/ramspeed
```

Figure 5.1: Example Test File

While creating the text file(Fig 5.1), and developing other similar features later discussed, particular attention was paid to make the functionality of the code portable between UNIX systems. Invoking the systems text editor was done in a way(Fig 5.2) that was POSIX compliant[14] to make sure that this method would work on other Linux systems.

```
"${EDITOR:-vi}" $testfile
```

Figure 5.2: POSIX Compliant method for invoking text editor

The next stage of the installation script(Fig 5.3) handles the installing of the tests specified in the test file. This is done by looping through the test file and invoking Phoronix's install test command. Here an issue was encountered. For each test specified, the installer would ask the user for confirmation that they wanted to install the test.

```
while IFS= read -r line
    do
    if  [[ $line == '#'* ]]; then
        continue
    fi

    yes | phoronix-test-suite install-test $line

done < $testfile
```

Figure 5.3: Phoronix test installation

This would become an issue for installing large amounts of tests and was also a redundant step from a user experience point of view, as the user had already confirmed the tests they wanted by making the test file. To solve this problem, the UNIX command yes was used to output an affirmative response to the install prompt, thereby removing any prompt to the user.

The final part of the installation script handled the configuration of the batch runs for Phoronix. The batch runs allow for running multiple tests with minimal user input but are not configured for the user by default. A pre-made user configuration file, .user-config.xml, was included in the project. Upon installation, the default configuration file is overwritten with the pre-made file, which is set to disable all unnecessary user input.

A portability error was discovered when making this feature. The installation.sh file is executed as root for previously discussed reasons. Commands executed are done so as the root user, which had the effect of making file paths that are relative to the user invalid. The solution(Fig 5.4) was to access the $USER variable as root to ensure that the file was moved to the right place.

```
cp "./.user-config.xml" "/home/${SUDO_USER:-${USER}}/.phoronix/user-config.xml"
```

Figure 5.4: Access $USER variable to make file paths user agnostic

5.2 Running Benchmarks

The next step of the implementation was to create a way for all tests that were installed previously to be run. This part of the framework was handled by the script run.sh in the project root.

A challenge became apparent with how Phoronix handles its test installations. Each test is designed to run by itself and be configured before it is run. This causes an issue where if multiple tests are chosen to run, a user would configure the first test, allow it to run, and the run would stop when the second test was reached as it then would need to be configured. This is a problem when a user could be specifying 1-N tests to be run, as the testing process would need to be watched at all times to not waste time with stalled runs.

This problem was solved by grouping tests together into test suites. A test suite is simply a group of tests that have been given an order to run and have been configured by the user before putting into use. A suite also has the advantage of being reusable, as in a user does not need to reconfigure tests to run the same way if they wish to run the same tests again. When a suite has been configured and saved, the framework will then make it run. This is done by listing all Phoronix's local test suites and selecting the most recent suite by the time created.

As the tests are kicked off, the user is prompted for input twice. They are first asked if they wish to use the time taken to run the benchmarks as an objective. This allows the user to do a multi-objective optimization even if they run a single test. They can compare the time taken to run the test against the output of the test. Secondly, the user is asked if they have an objective measured from outside of the framework that they wish to optimize. The idea behind this feature is to allow a user to specify metrics that may not be captured by the benchmarks. Such a metric may be the cost of the configuration(s) being optimized, or the energy costs of the configuration that have been measured by some external tool.

Lastly, some minor features were added to the run script to make the framework more convenient for the user. These features were implemented with Bash's getopts function. The purpose of this was to allow the user to specify certain functionality without having to run the whole framework every time. These options can be accessed by executing the run.sh script and using the appropriate flag. These features include:

- Quick access to writing the test file.
- A help page for using the framework.
- Listing the installed tests.
- Running a pre-existing test suite the user has made.
- Removing all installed tests.
- Running only the optimization program without running tests.

5.3 Pre-Processing of Data

By default, the output of the benchmarks is stored in the form of a HTML file. This created an issue whereby the desired data of the test runs is obscured by unneeded text surrounding it.

The initial approach taken to this problem was to try and parse out the relevant data with Bash scripts. This however proved to be very unreliable. The web page(Figure 5.5) can differ depending on the amount and types of tests run. The uncertainty of this output does not make it a guarantee that the correct details may be parsed each time. As a result, a different approach was taken.

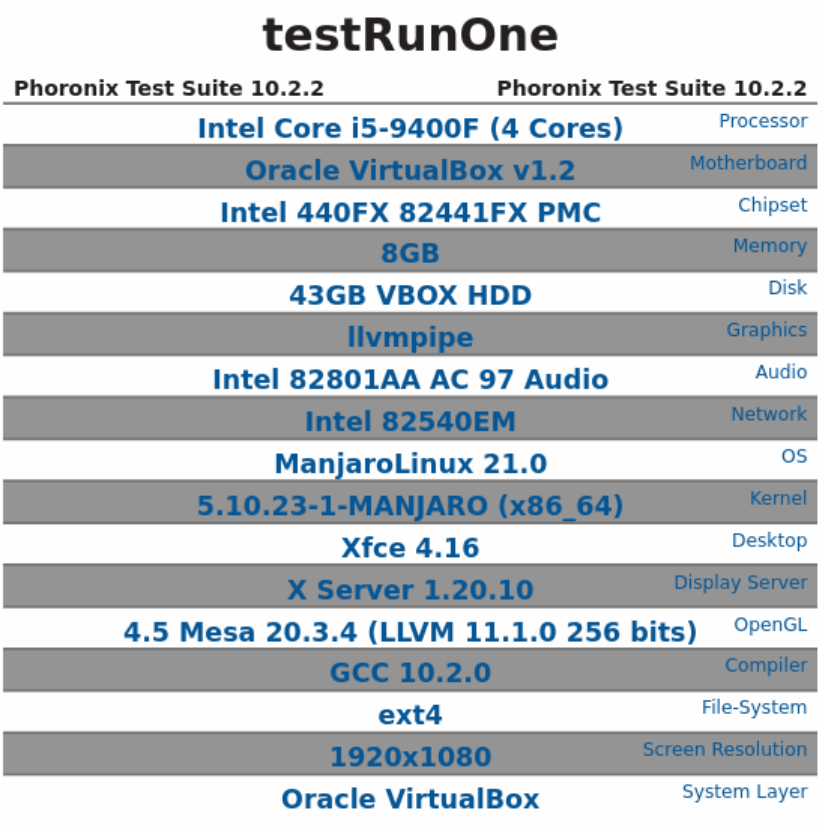

Figure 5.5: Phoronix Component Display

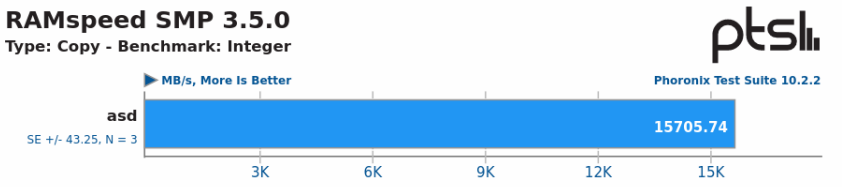

Figure 5.6: Phoronix Performance Graph

Phoronix has a method for converting the above data to a comma separated values(CSV) format (Fig 5.7). This format allowed for a much simpler and more reliable parsing. However, there was still unneeded information that should be removed before any optimizations could take place.

1	testRunOne		
2			
3			
4			
5	Processor		Intel Core i5-9400F (4 Cores)
6	Motherboard		Oracle VirtualBox v1.2
7	Chipset		Intel 440FX 82441FX PMC
8	Memory		8GB
9	Disk		43GB VBOX HDD
10	Graphics		llvmpipe
11	Audio		Intel 82801AA AC 97 Audio
12	Network		Intel 82540EM
13	OS		ManjaroLinux 21.0
14	Kernel		5.10.23-1-MANJARO (x86_64)
15	Desktop		Xfce 4.16
16	Display Server		X Server 1.20.10
17	OpenGL		4.5 Mesa 20.3.4 (LLVM 11.1.0 256 bits)
18	Compiler		GCC 10.2.0
19	File-System		ext4
20	Screen Resolution		1920x1080
21	System Layer		Oracle VirtualBox
22			
23			
24	RAMspeed SMP - Type: Copy - Benchmark: Integer (MB/s)	HIB	15705.74

Figure 5.7: Phoronix CSV Output

Creating CSV files from the test runs was done in a similar manner to how tests were made to run in the previous section. The most recent test result was obtained through the ls command (Fig 5.8), filtered by time to get the most recent file. The command was piped into the head command to give only one result. The file path had to be reduced to only the file name for the Phoronix method to work, so the basename Fig UNIX command was used to output only the file's name. Once the CSV file had been created, it could then be filtered easily with three commands.

```
result=$(basename  "$(ls -t "/home/${SUDO_USER:-${USER}}/
    .phoronix-test-suite/test-results/" | head -1)" )

    phoronix-test-suite result-file-to-csv $result
```

Figure 5.8: Creation of CSV file phoronix

First, the stream editor sed (Fig 5.9) was used to remove the first twenty-three rows. These rows are constant between test results and will always contain only the system's components. Second, the cut command was used to output only the third column of the CSV, where the pure data of the test results are always kept.The unit of measurement for each test was not deemed important to keep for the optimization as it is assumed the user knows the measurement as they specified the test, and they are not useful for the optimization algorithm. Lastly, sed is used again to replace the trailing newline character on each row with a comma so that each result is on the one row for the optimization stage of the framework. Any left over files are then cleaned up.

```
result=$(basename  "$(ls -t "/home/${SUDO_USER:-${USER}}/
.phoronix-test-suite/test-results/" | head -1)" )

if [ ! -e "./results" ];then
        mkdir "./results"
fi

phoronix-test-suite result-file-to-csv $result

file=$(ls -Art /home/${SUDO_USER:-${USER}}/*.csv | tail -n 1)

#Pre-process the csv
sed -i -e '1,23d' $file
cut --complement -f 1-2 -d, $file >> "./results/temp_results.csv"
sed -i -z 's/\n/,/g;s/,$/\n/' "./results/temp_results.csv"
echo $(cat "./results/temp_results.csv") >> "./results/test_results.csv"

rm "./results/temp_results.csv"

rm $file
echo "Result moved to results/"
```

Figure 5.9: Data Pre-Processing

5.4 Implementation of Multi-Objective Optimizations

With the data points prepared, the second to last stage of this project is to analyse the points input into the system, and then suggest the optimal configurations from the points given. This section of the framework is implemented in Python, and makes use of the library Numpy. Numpy was chosen as it contains a wide variety of functions for array manipulation and arithmetic, and works well for handling arrays with varying shapes.

Data created in the previous stages of the framework is collected and transformed into an N-Dimensional Numpy array. The array is then passed to a function called find_pareto(Fig.5.10). This function produces an array that has been organised into a Pareto Frontier. A Pareto frontier[15] is a set of points organised by Pareto Optimality, with the most optimal points being at the start of the array (0) and the least optimal at the end.

The function, which can be seen in Fig 5.10, first sorts the data by the sum of the coordinates in a decreasing order. This is done to prevent a point that has the largest coordinate sum in the set and that appears early in the data set from dominating all subsequent points.

A second array for undominated points is then created to be used as a Boolean mask for selecting points from the first array. A Boolean mask[16] is a matrix of Boolean values that uses a criterion to evaluate values in a collection. The objectives of the mask must match the dimensions of the array it is being used with. Values that pass the criterion can be mapped onto another array of the same size. In the case of this Boolean mask, the criterion is if a point is Pareto-dominated or not.

Pareto Dominance[15] is when a point A is better than or equal to a point B when their objectives are compared for Pareto Optimality. In the case of this function, each objective would be the

benchmarks the user ran for the current configuration. A point would be said to be dominant if it is more optimal for the user than the next.

The loop in the function iterates through each objective in the data set supplied. The Boolean mask is used to determine if the points in the current iteration are dominated or not. Points that are not dominated are kept to be compared in the next iteration. This has the effect of moving less optimal points into a higher index of the array and putting more optimal points at the lower indexes.

```
    data = np.genfromtxt('./results/test_results.csv', delimiter=',')

    def find_pareto(data):
      points = data[data.sum(1).argsort()[::1]]
      undominated = np.ones(points.shape[0], dtype=bool)

      for i in range(points.shape[0]):
        n = points.shape[0]

        if i >= n:
          break

        undominated[i+1:n] = (points[i+1:] >= points[i]).any(1)
        points = points[undominated[:n]]

  return points
```

Figure 5.10: Function for Finding Pareto Frontier

5.5 Output of data to User

The last part of the implementation was taking the points created in section 6.4 and creating a visual representation where possible. This was also done in Python, and used the library Matplotlib to produce two and three dimensional graphs.

Two functions were created to create the graphs. They have identical input, as they both take as arguments the points produced in Fig 5.10 and the data produced from the benchmarks. The reason two arrays are passed to these functions is so that the optimal points can be shown in contrast to the less optimal points.

The first function, that can be seen in Fig 5.11, creates a two-dimensional scatter plot. The optimal points are highlighted in red and the original points are highlighted in blue. Each axis of the graph is labelled according to their respective objective. The resulting graph is saved as a PNG file in the results directory.

```
def print2D(points, data):
      plt.scatter(data[:,0], data[:,1], c='b')
      plt.scatter(points[0], points[1], c='r')
      plt.xlabel('OBJ 1', fontsize=16)
      plt.ylabel('OBJ 2', fontsize=16)

      plt.savefig('./results/result.png')
```

Figure 5.11: Function for creating 2D graph

The second function, Fig 5.12, works similarly to the last function (Fig 5.10) but takes arrays with three objectives and creates scatter plots from them. Each objective of the array is mapped to an axis of the graph. This is repeated for the optimal points and benchmark data, again highlighting the optimal points in red and the less optimal in blue.

```
def print3d(points, data):
      fig = plt.figure()
      ax = fig.add_subplot(111, projection='3d')

      x = points[0]
      y = points[1]
      z = points[2]

      x1 = data[:,0]
      y1 = data[:,1]
      z1 = data[:,2]

      ax.scatter(x1,y1,z1, c='b')
      ax.scatter(x,y,z, c='r')

      ax.set_xlabel('OBJ 1', fontsize=16)
      ax.set_ylabel('OBJ 2', fontsize=16)
      ax.set_zlabel('OBJ 3', fontsize=16)

      plt.savefig('./results/result.png')
```

Figure 5.12: Function for creating 3D graph

Lastly, a simple series of if else checks (Fig 5.13) are used to determine the number of objectives in the arrays so that the correct function is called. If the objectives contained in the array are greater than three, no graph is created. Instead, the points are outputted to a CSV file. This also happens for two and three dimensional arrays along with their graphs.

```
if (points.shape[1] == 2):
  print2D(points, data)
elif (points.shape[1] == 3):
  print3D(points, data)
else :
  print(points)

np.savetxt("./results/optimal_result.csv", points, delimiter=",")
```

Figure 5.13: Function for creating 3D graph

Chapter 6: Testing and Evaluation of Results

In this section the evaluation and testing of the project will be discussed. Particular attention is paid to the methods used, some of which were inspired by the methods used in the Performance Evaluation of Linux Operating Systems[7] paper examined in 3.2.1. The main aspect to be evaluated is the effectiveness in which the framework chooses Pareto Optimal configurations from the configurations tested.

To test the framework, three different machines were obtained. Each machine had different hardware specifications, and each had a different installation of a Linux operating system. Of the three Linux distributions picked, two distributions (Manjaro and Garuda Linux) are based on Arch Linux. The third distribution, Linux Mint is based on Ubuntu. The machines used for the evaluation were as follows:

- A Lenovo 120s netbook.
- A Lenovo 330s laptop.
- A desktop PC running a virtual machine.

The details of these configurations can be seen in the Tables 6.1 - 6.3

Machine	Desktop VM
Processor	Intel Core i5-9400F (4cores)
Memory	8GB
Disk	43GB VBOX HDD
Graphics	lvmpipe
OS	ManjaroLinux 21.0
Kernel	5.10.23-1-MANJARO (x86_64)
Desktop	Xfce 4.16
Compiler	GCC 10.2.0
File-System	ext4

Table 6.1: Desktop VM

Machine	Lenovo 330s
Processor	Intel Core i5-8205U
Memory	8GB
Disk	256GB SK hynix HFS256G39TNF-N3A
Graphics	Intel UHD 620 3GB(1100MHz)
OS	Garuda Soaring
Kernel	5.11.10-zen1-1-zen (x86_64)
Desktop	KDE Plasma 5.21.5
Compiler	GCC 10.2.0 + Clang 11.1.0
File-System	btrfs

Table 6.2: Lenovo 330s

Machine	Lenovo 120s
Processor	Intel Celeron N3350
Memory	4GB
Disk	31GB DF4032A
Graphics	Intel HD 500 3GB
OS	Linux Mint 20.1
Kernel	5.4.0-58-generic (x86_674)
Desktop	Xfce 4.14
Compiler	GCC 9.0.0
File-System	ext4

Table 6.3: Lenovo 120s

Three sets of tests were run on the machines. The first set involved two benchmarks, each acting as their own objective. The second set used three benchmarks, and the third test used five benchmarks. The benchmarks that were chosen gave a general overview of performance. The aims of these tests were to see how well the framework ran on each configuration and if the results obtained from comparing the configurations were accurate.

Benchmarks used:

- Scimark2: Measures computational kernels[17]
- Ramspeed: Measures the speed of the RAM in the configuration.[18]
- Fio: A multithreaded I/O testing tool[19].
- Cachebench: Tests the architecture of the memory subsystem[20].
- Compress-gzip: Tests the speed at which the configuration can compress files[21]

For the bi-objective tests, Cachebench and Compress-gzip were be used. For the tri-objective tests, Scimark2, Ramspeed and Fio were used. The tests with five objectives used all the benchmarks. This was done for two reasons. The first was to examine the framework's accuracy when handling increasing numbers of objective. Secondly, the values of the units of measurement for each benchmark vary in whether they are better in larger or smaller values. For example, Cachebench is measured in MB/s, whereas Compress-gzip is measured in seconds. A larger Cachebench score is better than a lower score, but the opposite is true for Compress-gzip. This tests the frameworks ability to choose the optimal result if the worth of the values vary.

The accuracy of each test is determined by Phoronix's own statistical accuracy handler[13]. This handler detects if a test run's result is landing outside of a predefined standard deviation, and will start additional runs as necessary to ensure accuracy.

6.1 Results

Machine	Gzip	Cachebench
120s	90.293	16005.87
330s	46.849	38561.29
VM	44.24	43588.82

Table 6.4: Results of 2D tests

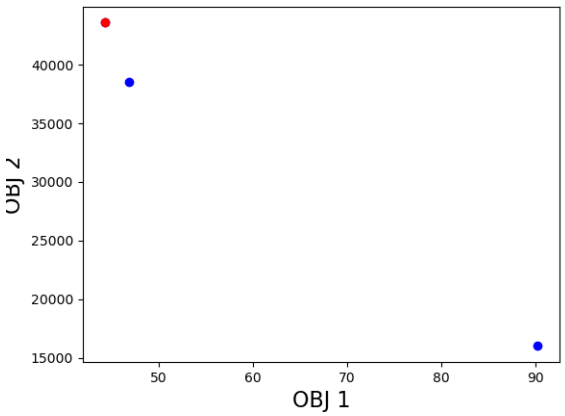

Figure 6.1: Scatterplot of 2D test run

Machine	Scimark2	Fio	Ramspeed
120s	182.49	1410	16229.64
330s	149.99	1037	16103.72
VM	182.99	1410	16229.64

Table 6.5: Results of 3D tests

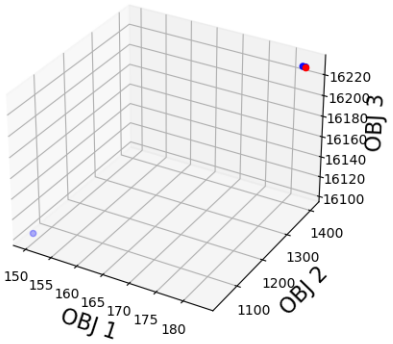

Figure 6.2: Scatterplot of 3D test run

Machine	Fio	Ramspeed	Cachebench	Scimark	Gzip
120s	741	9447.21	16010.09	35.37	93.41
330s	1044	16075.34	38518.65	150.62	47.87
VM	1424	15724.13	43386.17	168.23	44.88

Table 6.6: Results of 5D tests

The results seen in Tables 6.4 -6.6 are displayed how they were outputted by the optimization algorithm. They each create a Pareto Frontier, with the least optimal result displayed at the top of the table, and the most optimal at the bottom. These results are further explained and discussed in the following sections.

6.2 Evaluation

The evaluation written here will be primarily concerned with what this project has achieved relative to the goals set out in the Project Specification. As there is no framework that is similar to this currently available, comparison of certain aspects of the project is difficult. The framework's inputs, processes and outputs will be discussed in turn, and where comparisons can be made they will be evaluated.

6.2.1 Inputs

The input of the framework was tested by specifying the tests which the benchmarking suite was to install when doing the tests in section 6.1 . In the case of each machine that the framework was tested on, each test was installed and run from within the framework without error. Multiple tests could be specified in the test file and were installed correctly. In the test runs, it was needed to install additional tests after the bi-objective test runs had been completed. This was done by using the -i flag in the run script to install additional tests. These tests were installed correctly. When running the five-objective test run made up of the previously ran tests, all tests were able to be configured to run together without issue.

An error did occur when running the installation on the machine with Linux Mint. The installation script encountered an issue when installing Phoronix. The benchmarking suite failed to install correctly and had to be manually installed by the user. This did not occur on the Manjaro and Garuda installations.

6.2.2 Processes

When evaluating performance of the optimization algorithm from section 5.4, the output of the process and the execution times were examined. The tables 6.4 - 6.6 show that each set of results were organised correctly into a Pareto frontier, as the virtual machine performed the best relative to all tests and was placed at the bottom (the most optimal) in the case of these tests.

A direct comparison to other Pareto Optimization algorithms is difficult as other examples of these algorithms are contained in large libraries of code with different use cases. One such example would be the NSGA algorithms found in the Pymoo Multi-objective library[22]. These are large programs capable of dealing with large data sets and employ genetic algorithms to find their solutions. The algorithm in this framework is much smaller in scale and designed to work for a specific use case.

The execution time of the algorithm used by the framework did not increase significantly with each increasing objective either. The results of the tests done in the previous section were used to test the execution time for two, three, and five object configurations. These results were passed through the program ten times each and measured with the time Unix command to get an average performance. These were the results:

- 2 Objective Performance average : 0.431.3 seconds
- 3 Objective Performance average : 0.453.5 seconds
- 5 Objective Performance average : 0.435.6 seconds

6.2.3 Outputs

The graphical outputs (Fig 6.1 and Fig 6.2) are two- and three-dimensional scatter plots. In each plot, the blue dots represent a configuration while the red dots represent the Pareto optimal configuration. In the two dimensional graphs, the axis OBJ1 represents the compress-gzip results and is measured in seconds. The OBJ2 axis represents the Cachebench scores which are measured in MB/s.

The results for the tri-objective tests are displayed similarly. The Pareto optimal configuration is shown in red and all other configurations are shown in blue. OBJ1 is the measure of Scimark2 results, which are measured in MFLOPS. OBJ2 measures the results of fio in MB/s. Lastly OBJ3 was for Ramspeed and was also measured in MB/s.

The test with five objectives was outputted as a CSV file, with the Pareto optimal result shown at the bottom of the file.

The plots created here clearly show the most optimal result from the tests run. While the results may seem obvious in this case as there are only three configurations being tested, with larger amounts of configurations it would be harder to tell immediately what the optimal configuration is. The red dot would be visible among all other configurations and immediately highlight the optimal configuration. When the graphed results are used with the CSV output, a clear image of how the configurations performed can be seen.

Chapter 7: **Future Works and Conclusion**

This framework could be just the beginning of a bigger project. The features described in chapter 5 can be improved and expanded upon to further the projects' goal of user specified testing.

An area improvement for this project would be the displaying of the optimal configurations. More detailed graphs could be implemented that label each point with their configuration for clarity, and more clearly display the Pareto frontier could be employed. Furthermore, the graphs could be expanded to illustrate the performance of multi-objective data past the third dimension. In Dipanjan Sarkar's article on Strategies for Effective Data Visualization[23], he describes several methods for displaying multidimensional data. Of note was his example of Parallel Coordinates for visualising data. He shows an implementation in Python that could be adapted for the purposes of this project and could effectively contrast performance of different configurations with large amounts of objectives.

An area that could be expanded on is what the framework measures. For the purposes of this project, only Phoronix was implemented. With more time additional benchmarks, such as those mentioned in section 3 could be added to the framework. The performance of specific applications could also be measured. The purpose of doing this would be to increase the level at which a user could specify their testing. For example, a user who wishes to pick the optimal configuration for computer gaming, versus a user who wants the best configuration for a particular compiler will both want to measure the performance of certain applications, but in different ways. Testing applications that run in the background and how they affect the performance of a configuration would be an ideal area for the optimization present in this framework.

Lastly, a graphical user interface could be developed to expand the framework beyond just the Unix command line interface. A GUI can offer a more intuitive user experience for user's not familiar with CLI and therefore expand the user base of the framework. Additionally, if combined with the graphical improvements mentioned previously, an implementation like Grafana[10], examined in section 3.2.3 could be used to make the data produced more interactable. Features like filtering data by the performance of a particular objective, or looking at the performance of a subset of the configurations would increase the overall user experience.

7.1 Conclusion

This project aimed to create an open framework for finding the optimal configuration among a series of configurations specified by a user. This aim was achieved by allowing the user to specify which tests they wished to install and run from a benchmarking suite, then allowing the user to configure those tests to their needs. The data which the test runs produced was then analysed to display the optimal configuration from the results.

From the evaluation of this project, these goals have been reached and the advanced goal of displaying the Pareto Optimal configuration in a graphical form was also achieved.

From researching this project, I believe that a framework such as this could contribute to the field. As computers become more common place the need for testing and evaluating systems increases. There are several benchmarking suites that exist but creating a means to test and evaluate a machine in a simple way could encourage more people into Computer Science. All software used in this project was free to use and open source, so the barrier to entry is quite low.

The project can be viewed on the following Gitlab Repo. https://csgitlab.ucd.ie/james.kirwan/final-year-project

Chapter 8: Addendum

8.1

Project Workplan

Note, this section was written in December of 2020. The workplan changed as features were implemented. Virtualisation was an element not included in the end project.

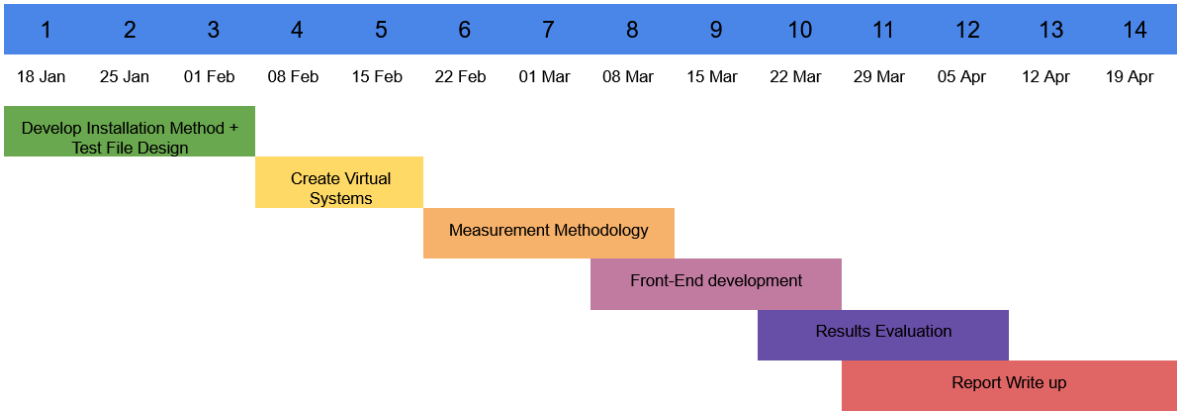

Figure 8.1: Project Work Plan Gantt Chart

8.2 Installation Method and Test File Design

A. This period will focus on developing scripts to run the framework. Benchmarks will be installed and run, and any directory structures will be created. Their results will also be collected. A directory tree will be developed to track where each element of the project will be stored.

B. Additionally this period will be used to develop a test file which the user can input details pertaining to what software they want installed and which metrics they want tested.

8.3 Create Virtual Systems

C. This section of work will involve implementing Qemu into the project. Linux system images will be created with the work from the previous section on them. The system will then be tested from the initial stage of inputting a test file to running the benchmarks to provide data for the next section.

8.4 Measurement Methodology

D. In this period, the focus will be on creating metrics for result comparison between the different test runs. A points system will be developed to mark how well a system performs. These metrics will be designed with the front end in mind so that they can be easily graphed and displayed in the next section. Additionally these metrics will also be designed to fit an optimization model.

8.5 Front-End Development

E. This section will involve writing code to generate graphs and web pages for viewing results at the end of a testing cycle. These results will be split up into singular results per system tested and comparative results between all systems.

F. Additionally, this section will be used for user specified output. A means of specifying what output to display will be created. Particular attention will be paid to usability and to what attributes may be displayed.

8.6 Results Evaluation

G. This period of time will see a review of results gathered so far. Initially the usage of the of the whole framework will be scrutinised. A focus on how the framework differs from other benchmark suites will be taken.

H. Upon the completion of the front-end, the resulting data can be evaluated. The framework will be tested on at least two different computers with the same set of Linux distributions. The goals of this evaluation will be to see how do the student made metrics fair. Do they show a clear difference in performance between systems. Are the results easy to view. How much choice does a user have when viewing their test results.

8.7 Report Write Up

I. Lastly, the student will begin to write up the last stages of the report. Throughout the project, sections of the report will be written and diagrams or other relevant works will be created. This last section is for writing up any unfinished sections, and ensuring the paper is understandable and of good quality.

Acknowledgements

I would like to thank my supervisor, Professor Alexey Lastovetsky, for the invaluable advice he has given and the guidance he has provided.

I would like to give special thanks to Saoirse Houlihan and William Burke for provided feedback and suggestions throughout the development of this project.

I would also like to thank my family for the support they have given to me throughout my education.

I want to acknowledge the user Peter from the linked StackOverflow thread as the optimization function used in this project was inspired by their explanation of Pareto programming. https://stackoverflow.com/questions/32791911/fast-calculation-of-pareto-front-in-python

Bibliography

1. *Distrowatch.com* 2020. https://distrowatch.com/.
2. Linux Operating System Market Share Covid-19 Impact Analysis By Distribution, By End Use, and By Regional Forecast. https://www.fortunebusinessinsights.com/linux-operating-system-market-103037 (2020).
3. *Phoronix Test Suite - Linux Testing and Benchmarking Platform* 2020. https://www.phoronix-test-suite.com/?k=home.
4. Kopytov, A. Sysbench manual. *MySQL AB* (2012).
5. Kelly, L. *BYTE UnixBench* 2015. https://github.com/kdlucas/byte-unixbench.
6. *Openbenchmarking.org - cross-platform, open-source automated benchmarking platform* 2020. https://openbenchmarking.org/.
7. Boras, M., Balen, J. & Vdovjak, K. *Performance Evaluation of Linux Operating Systems* in *2020 International Conference on Smart Systems and Technologies (SST)* (2020), 115–120.
8. Chiandussi, G., Codegone, M., Ferrero, S. & Varesio, F. Comparison of multi-objective optimization methodologies for engineering applications. *Computers Mathematics with Applications*. https://reader.elsevier.com/reader/sd/pii/S0898122111010406?token=16ECF94C889FB8C58D2626E1EA169050348356D91C1F4587413DC5398C153CFAB41ED286F63A51E196C3B8 (2011).
9. Ingham, S. Pareto-Optimality. https://www.britannica.com/topic/Pareto-optimality (2019).
10. Labs, G. *Grafana* 2020. https://grafana.com/grafana/.
11. Labs, G. Grafana Documentation. https://grafana.com/docs/grafana/latest/ (2020).
12. S. Suffian, K. M. *Nonconventional and Vernacular Construction Materials, Second edition* chap. 3 (Elsevier, 2016).
13. *Phoronix Test Suite Features* 2020. https://www.phoronix-test-suite.com/?k=features.
14. Kenlon, S. What is POSIX? Richard Stallman explains. https://opensource.com/article/19/7/what-posix-richard-stallman-explains (2019).
15. Costa, N. R. & Lourenco, J. A. *Transactions on Engineering Technologies* chap. 27 (Springer, 2014).
16. Agarwal, Y. *Broadcasting and Boolean Mask* 2020. https://medium.com/@er.26yashiagarwal/broadcasting-and-boolean-mask-ab1ddae2da76.
17. Pozo, R. & Miller, B. *Scimark 2.0* 2004. https://math.nist.gov/scimark2/.
18. Hollander, R. M. & Bolotoff, P. V. *Ramspeed, a cache and memory benchmarking tool* 2018. https://github.com/cruvolo/ramspeed-smp.
19. Axboe, J. *fio - Flexible I/O Tester rev.3.26* 2021. https://fio.readthedocs.io/en/latest/fio_doc.html.
20. Mucci, P. J. *Cachebench* 2009. http://icl.cs.utk.edu/llcbench/cachebench.html.
21. Gailly, J.-l. & Adler, M. *Gzip Compression* http://www.gzip.org/.
22. Blank, J. *Pymoo Github* 2021. https://github.com/msu-coinlab/pymoo.

23. Sarkar, D. *The Art of Effective Visualization of Multi-dimensional Data* 2018. https://towardsdatascience.com/the-art-of-effective-visualization-of-multi-dimensional-data-6c7202990c57.

List of Figures