ACIT5900

MASTER THESIS

in

Applied Computer and Information Technology (ACIT)

August 2024

Cloud-based Services and Operations

Automatic Detection of Abnormal Behavior in Virtual Machines with Consideration of Policy-Based Response

Henrik André Ruud

Department of Computer Science

Faculty of Technology, Art and Design

OSLOMET

0.1 Preface

In an era where an increasing amount of internet services resides on large centralized cloud centers, keeping them continuously operation has never been more important. This makes it very important to have good, and working methods of anomaly detection so any and all downtime can be avoided. This report aims to look at what might cause a large service provider downtime, and then investigate how it could have been avoided. The purpose is to investigate a specific type of anomaly detection called the leap detection test which is based on the chi-squared test, and then discuss how it could potentially pair with a policy-based decision-making tool.

This topic was chosen as the area of research since it interests me and it would allow for the development and testing of an anomaly detection prototype.

I am grateful to the Oslo Metropolitan University where i have studied and learned much. I am especially grateful to my supervisor Kyrre Begnum who have supported me and given me valuable insight while i have been working on this project. Lastly i want to thank Hedda Marie Westlin for the collaboration in providing me with the dataset used in this project.

Henrik André Ruud

15.08.2024 Slemmestad, Asker.

0.2 Abstract

This study tries to see if it is possible to design and develop an anomaly detection algorithm by using an algorithm based on the chi-squared test. It also looks at how that potential anomaly detection algorithm could be paired with a policy-based decision-making system. The goal of this research is to try and detect crashes and abnormal behaviors in virtual machines without having access to any personal or sensitive data, and doing so only by viewing the CPU usage. The second goal is to investigate how this could lessen the burden on the continuously growing cloud infrastructure by discussing how the this anomaly detection algorithm could work together with a policy-based decision-making system to free up resources and manpower. The scope of the project is to only work with this one algorithm to find out how it performs specifically.

To do this, a prototype which uses the leap detection algorithm which is the algorithm in question was designed and developed. It was used on data from real virtual machines. It underwent three different iteration based on test results before reaching its final form. At that point, it was made clear that it was not suited to reliably detect crashes but instead had potential to detect sudden increases in CPU usage. There was no way for the algorithm do reliably distinguish a sudden spike in CPU usage from an anomaly or crash. It was therefore suggested other use-cases such as monitoring CPU usage patterns and warning about sudden spikes in usage. It was then suggested that different algorithms should be explored for further research into the subject.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the concept of cloud infrastructure were introduced, it has steadily grown till the point where it is at now, responsible for hosting a large amount of the service on you can find on the web. This has however not been without challenges and difficulties along the way. As clouds grow, the infrastructure and management necessary grows with it. Since the change to cloud infrastructure is still an ongoing process the clouds will only grow larger leading to even greater challenges managing and operating these clouds. The challenge grows proportionately with the size of the cloud.

What makes managing and operating a cloud infrastructure challenging is its dynamic nature and evolving demands. This makes managing the cloud an exercise of uncertainty. This can be credited to unpredictable fluctuations in user traffic, varying application requirements and the rapid pace of technological advancement in the field. Another challenge is also that it is the customer itself who control what they have on the cloud whether that is outdated software or new software with undiscovered faults.

Since the customer controls what they have on the cloud, they in turn have influence over how their virtual machines behave and what resources they take up. The problem here arises when the customer manages their virtual machines in a way that might lead to unexpected behaviours. Such unexpected behaviours can be abnormal behavior or system errors caused by what the customers do with their virtual machine. If such errors appear on the virtual machine in the cloud, it could lock out resources which could have been used by other customers. One of the reasons this is a big problem is due to the nature of overbooking. Overbooking usually works out fine since a virtual machine rarely uses all available resources, which means that when they suddenly do, they hinder others from using those overbooked resources. This leads to a poor user experience for the customer who's machine crashes and for other potential customers who wanted to use the resources being occupied.

To combat this challenge, the cloud provider responsible for managing the cloud would ideally take action and do something about the virtual machines who have ran into this abnormal or unexpected behavior.

The provider could reboot the machine or attempt to address the problem in some other suitable way so that the user experience don't get tarnished and the resources are freed.

The first challenge that needs to be dealt with to solve this problem is that the virtual machines experiencing issues need to be identified. This is because if we want to take action to solve the issue, we first need to be made aware of it. Since the cloud provider don't have access to whats on the virtual machine, the identification must be based on the resources the machine uses and how they are used. This identification process is a huge task which means that it must be done automatically for it to be a viable step in solving the challenge.

Once a virtual machine has been identified as having problems, a response is needed. The response will need to follow policy based guidelines where the most suitable response would depend on several factors. These factors will depend on the state of the virtual machine and it's resources. Furthermore this response needs to be automated so that it happens at the moment the problem has been automatically identified. Automated detection of faulty virtual machines together with automated policy-based response will be the backbone of the cloud infrastructure.

1.1 Problem statement

The problem statement that emerges from the introduction is as following: *Design and develop a framework which can automatically detect abnormal behavior in virtual machines, such as system faults and resource hogging, with a discussion on potential policy-based responses.*

The *framework* in our context is a set of tools which *automatically*, with minimal input from an admin *detect abnormal behavior* such as system faults and *resource hogging*. *Abnormal behaviors* will be detected by monitoring the machine's utilization of it's available hardware resources. The goal is to detect *resources hogging* which is when a machine uses a large amount of resources for an *abnormal* amount of time. If abnormal behavior is detected, a *policy-based response* will will follow automatically. The policy will dictate which response should be taken. The *response* should be *automated* so it requires minimal input from an administrator.

1.2 Document overview

This thesis is organized into the following chapters:

- **Background:** Introduces background information and the history of cloud computing. It then goes into related research on the field of anomaly detection and policy-based decision-making
- **Methodology:**Details the scope of the project and explains design chooses and reasons behind them. Also discusses how to evaluate the eventual product of the project

- **Results:** Goes though the development of the prototype before proceeding to the test results of the algorithm.
- **Discussion:** Discusses the results of the the prototype and algorithm. Takes into consideration its potential and shortcomings, as well as discussing further research directions.
- **Conclusion:** Concludes the thesis by summarizing the key findings and contributions.

Chapter 2

Background

In this chapter, we are going to take a look at the research field surrounding the problem described in the problem statement. We are going to look at related research to see how far along the field is and get an understanding of where the current research focus is. The first point is to look at how we arrived at where we are at now when it comes to cloud infrastructure. We will also look at how cloud services are operated and the difficulties which arises when the cloud grows in size. The second point is to look at different approaches to anomaly detection and how successful they are. Then comes the field of automated decision-making. The goal is to investigate different types of automated decision making methods to get a better understanding of where the research is at.

2.1 Operating cloud services

The cloud infrastructure is an ever-evolving field that keeps on growing and simultaneously evolves to keep in check the growing demand for its services. Managing a cloud service poses many challenges that must be overcome to successfully keep the service operational. Some of these challenges are managing resources and making decisions on what to do when anomalies appear. Before we look closer at this, a brief introduction to the history of the cloud computing will be presented.

2.1.1 History of cloud computing

The concept of cloud computing can be dated back to visionary concepts in the 1960s. It began to take shape with the advent of models such as grid computing and utility computing, which can be said to have combined into the type of cloud computing we have today. Utility computing is a model where computing resources are provided over the internet such as electricity. It is based on the pay and use model where you only pay for the resources you use and the time you use them. This business was manly directed at large organizations and banks which would pay to use the providers' servers before having in house computers became the norm("Utility computing", 2024). Grid computing however is not a model on how to provide and monetize computing power, but instead is model on how to combine computing power. It involves coor-

dinating multiple computers to work towards a single goal allowing for the harnessing of power from multiple computers for one goal("Grid computing", 2024). These concepts date back to the 1960s(Surbiryala & Rong, 2019) but did not see widespread adoption until the 1990s. We did also not call the models the same as we do today. The most common name for utility computing in the early days were time sharing whereas grid computing did not get its name until 1999 from the seminal work, "The Grid: Blueprint for a new computing infrastructure" ("Grid computing", 2024). These early frameworks laid the groundwork for modern cloud computing by emphasizing the efficient sharing and distribution of computing resources.

The introduction of the concept Software as a Service (SaaS), which began appearing in the late 1990s with the emergence of companies like Salesforce and NetSuite, allowed services to be accessed and utilized over internet. Some early examples of Software as a Service are tools such as email, customer relationship management, and project management software("What is SaaS?", n.d.). Salesforce started in 1999 and offered a web-based customer relationship management solution to other companies. NetSuite launched a few years before in 1995 and provided accounting software and enterprise resource planning software. This was then expanded on by the launch of platforms like Amazon Web service (AWS) and Google's AppEngine in the mid 2000s which offered Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) continuing the evolution towards the type of cloud computing we have today. Platform as a service is to provide a platform allowing customers to develop, run, and manage applications without dealing with the complexities of infrastructure management, while providing infrastructure as a service is to provide virtualized computing resources over the internet.

The late 2000s and early 2010s were good years for new and innovative releases in the cloud industry. The focus during this period in cloud computing was availability, ease of use and scalability, which is still very much in focus today. It lead to the development of infrastructure and software used in hosting and controlling clouds. Some of the more famous releases in this time period were initiatives like the OpenStack project, Apache CloudStack and Eucalyptus. These three projects have been staples in the cloud computing scene.

The OpenStack project launched in 2010 as a free open standard cloud computing platform, which it still is to this day. It emerged as a collaborative effort between Rackspace Techology, then known as Rackspace Hosting and NASA. The goal of this collaborative effort was to produce an open source cloud computing platform that would be simple to implement and allow for easy and massive scalability. This in turn lead to cloud computing tools and services becoming accessible to a broader range of users and thus fostered innovation in cloud infrastructure. Furthermore, the integration of cloud computing with emerging trends such as mobile computing, Big Data analytics, gaming, and the Internet of Things (IoT) broadened the scope of cloud applications and fueled its widespread adoption across diverse domains("OpenStack", 2024).

Eucalyptus was released in 2008 as a software to manage cloud instances and entered in a partnership

with Amazon Web Services (AWS) in 2012. It was a widely used software for building AWS compatible cloud computing environments. Its name was an acronym for Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems which is quite a mouthful. It saw many years of service but was last updated in 2018("Eucalyptus (software)", 2024). Apache CloudStack began in 2008 as well, but at that time it was known as VMOps. In 2010, the company behind CloudStack changed its name to Cloud.com and released much of the CloudStack under the GNU General Public License Version 3 (GPLv3). The GNU General Public License("The GNU General Public License v3.0 - GNU Project - Free Software Foundation", n.d.) is a copyleft license intended to guarantee the users freedom to share and change a program so that it remains as a free software for all users. In 2012 after having changed owners, Apache CloudStack had its first major release. It has since been developed further and now shares its market with OpenStack("History of Apache CloudStack — Apache CloudStack", n.d.).

However, it's rapid expansion and widespread adoption has lead cloud computing to encounter formidable challenges in the realm of management and security. Ensuring the confidentiality, integrity, and availability of data stored in the cloud environment has become a concern requiring apt management and security. Furthermore, the multitude of use cases and users on the cloud has made keeping the cloud healthy a challenging process.

The evolution of cloud computing from its earliest stages to its present state represents a multitude of technological innovations which have come together to form the cloud infrastructure we have today. Although cloud computing today is the cornerstone of modern IT infrastructure, it is not perfect and will continue to evolve and expand. This leads us to the challenge this study is trying address which is the the challenge of managing a cloud environment which gets exponentially harder as its user base grows. A possible solution to accommodate this challenge is to utilize anomaly detection and policy-based decision-making to negate the challenges which arises.

2.1.2 Challenges exponential with size

Originally praised for its scalability and cost-effectiveness, cloud computing introduces a couple of new challenges along with it. Before this, we need to address the concept of overbooking which have been mentioned in the introduction. Overbooking is a common occurrence in cloud computing and is in fact the standard approach when dealing with cloud environments. This works by assuming that a users will very rarely use all the resources they paid for and if they do, it won't be for an extended time. This approach generally works as the assumptions are correct and they can effectively sell more resources than they have. The problem here arises when virtual machines do consume all resources available and do so for an extended time period.

2.2 Anomaly detection

Imagine running a cloud computing service where you provide your customers with virtualized computers where they can run whatever they want. You also utilize the overbooking principle as that has been proven to work in this business. This means that you overbook your resources and have little to no control over what your customers do with their virtual machines. This poses no problem in an ideal scenario, but as we know, there is always something that does not work as intended.

In this scenario, there is a virtual machine which keeps on crashing and simultaneously using 100% of the available resources. The machines keeps on doing this while going unnoticed. This creates a problem for the cloud provider and the other customers. Since the machine is hugging all available resources continuously it blocks other customers from using those resources. It also creates a problem for the customer who pays for the machine as they might notice a problem but not necessarily realize that it is their VM that crashes. The provider might also wonder why some of their resources are always occupied. The problem here is that the crashes are going unnoticed and would require a lot of additional work out of the administrator to finally realize where the issue is.

This is one of the challenges that administrators of cloud services are faced with. To respond to these challenges, different anomaly detection techniques tailored to the cloud environment have been developed. These techniques aim to leverage machine learning and statistical analysis together with special algorithms to detect when a virtual machine's behavior deviates from its expected behavior. The goal is to find which virtual machines experience abnormal behavior so that we can take further action regarding those machines.

This section will look at what anomaly detection methods there are out there. It will look at what already exists and explore briefly what they do and how they work. The aim is to look at detection methods that mainly use resource based detection, as the aim it to detect abnormal behavior without actually looking at what's inside the virtual machines in question.

2.2.1 Resource based detection

Resource based detection is not a new area of research, but it is not one that receives a lot of attention. Two papers which discusses and develops anomaly detection by looking at resource usage are the papers by Liu, Zhang and Xu (2018) named *An anomaly detector deployment awareness detection framework based on multi-dimensional resources balancing in cloud platform*, and the paper by Islam and Miranskyy (2020) named *Anomaly detection in cloud components.* Both papers focuses on anomaly detection but uses the metric resource utilization a little differently

While the first work by Liu, Zhang and Xu (2018) uses resource metrics and runtime performance metrics for understanding and modeling the environment it observes, the paper is more focused on resource

usage for optimized deployment of the anomaly detection method. The paper is about developing a framework they call DDAF which focuses on optimizing the deployment of the anomaly detection framework. This optimized deployment is where resource utilization metrics plays a larger role(Liu et al., 2018)

The other paper mentioned by Islam and Miranskyy (2020) however, focuses on anomaly detection by analyzing resource utilization metrics such as CPU usage, memory and network usage. They develop something they call the GRU-based anomaly detection model which stands for Gated Recurrent Unit based. The performance of the GRU-based anomaly detection model is good and received a score of 59,8 in Numenta Anomaly Benchmark(Islam & Miranskyy, 2020).

The first example of papers discussed here does not seem to apply to this project's scenario as the focus of the paper is optimized deployment instead of actual optimized detection. The second example however uses resource metrics during the anomaly detection. The problem here is that they have access to multiple metrics and use all of them during the detection. For this project with the algorithm chosen, only one metric will be look at and that is the CPU usage. This is to some extent done to avoid being able to see what happens inside the virtual machines observed. The goal is to be able to detect anomalies without having access to more than just the CPU utilization.

2.3 Policy-based decision making

Another area of importance in the problem statement is policy-based decision-making. This field represents systems which based on predefined rules and information provided makes decisions. The decisions are often made fully automatic by the decision making system, or require as little input from an administrator as possible. Research in this field goes by many names such as automated decision-making mechanism and multicriteria decision-making, but we will in this paper refer to this concept as policy-based decision-making.

In the paper *Big data analytics for event detection in the IoT-multicriteria approach* by Granat, Batalla, Mavromoustakis and Mastorakis (2019), they discuss the problem of multicriteria decision-making in detecting anomalies in the internet of things(IoT). They discuss multiple approaches to handling and reacting to events caused by anomalies. For one of the approaches they introduce something they call ESF which is short for event strength function. This approach allows the assigning of a strength level to the anomaly based on criteria set. Different alarms and actions can then be taken based on the strength level set (Granat, Janusz et al., 2019).

Another paper called *Evaluating the performance of cloud services: a fuzzy multicriteria group decision making approach* by Wibowo and Deng (2016) delves into multicriteria decision-making by providing a structured approach to evaluate and select the best option among multiple alternatives. In situations where the decision-makers face uncertainty or imprecision, fuzzy logic is used(Wibowo & Deng, 2016). Since fuzzy logic

allows for a degree of truth instead of strictly making it true or false, it makes it possible to still make a decision when it would have required intervention otherwise.

2.4 Leap detection test

The leap detection test which this projects solution is based on is an adaptation of the mathematical chi-squared test. The chi-squared test is a statistical hypothesis test used to compare an actual value to a set of known values. It is usually used to find out whether a new value fits in with population of values already observed. It can be used to calculate standard deviation.

To give an example, imagine having a collection or population of data on children's weight. It is then possible to use the chi-squared test to find out if the a new child's weight matches those previously observed. It is done by providing the sum of the weights, the size of the collection/population and the new weight. This will give out a value which will be named chi-value in this report. This chi-value can then be compared to another value which will be named chi-limit to determine whether the new weight matches the population. If the chi-value is higher than the chi-limit, it is statistically out of the ordinary. This test will be influenced by both the population size and the chi-limit.

The chi-squared test will be implemented into the code. The population will be previous CPU usage values whereas the chi-limit will be set to try and optimize the test for this project's specific scenario. The way the chi-value will be calculated is shown below:

$$\chi = \sqrt{\frac{(\text{sum} - \text{Memory\_Length} \times \text{diff})^2}{\text{Memory\_length} \times (\text{Memory\_Length} + 1) \times \text{avg}}}$$

Chapter 3

Methodology

This chapter will present how the project has been approached. It will go into detail on how decisions were made and technologies chosen. It will discuss the desired outcome of the project and how that outcome might be reached. It will also discuss the design of the prototype and at last how the prototype and algorithm will be evaluated.

3.1 Problem Statement Recap

The management of cloud infrastructure presents significant challenges due to its dynamic nature and evolving demands. Unpredictable fluctuations in user traffic, varying application requirements, and customer control over virtual machines (VMs) contribute to the complexity of cloud management. One of the primary challenges arises from unexpected behaviors exhibited by VMs, such as system faults and resource hogging. These anomalies can lead to service disruptions, hinder resource allocation, and degrade user experience. Addressing these challenges requires proactive measures to detect and respond to abnormal VM behaviors in a timely manner.

Effectively managing abnormal VM behaviors is crucial for ensuring the stability and efficiency of cloud infrastructure. Failure to promptly identify and address anomalies can result in degraded performance, lack of resources, and increased operational costs. Moreover, in overbooked environments, abnormal VM behaviors can lead to resource scarcity and compromise service availability, leading to a poor user experience for the customers.

Existing approaches to anomaly detection in cloud environments often rely on manual intervention or lack automated response mechanisms. This gap underscores the need for a more efficient and proactive solution that can automatically detect abnormal virtual machine behaviors and apply policy-based responses in real-time. By addressing this gap, the proposed framework aims to overcome the limitations of current solutions and enhance the resilience of cloud infrastructure.

The objective of this research is to design and develop a framework that can automatically detect abnormal VM behaviors and apply policy-based responses to mitigate their impact. By leveraging automated detection mechanisms and policy-based decision-making, the framework aims to improve the stability, efficiency, and user experience of cloud infrastructure. This objective aligns with the broader goal of optimizing resource utilization and ensuring the reliability of cloud services in dynamic and evolving environments.

3.2 Desired Outcome

The goal and desired outcome of this project is to have looked at and tested at least one possible solution to the problem mentioned above and in the problem statement. We want to be able to detect an anomaly based on information which consists of a virtual machine's resources utilization over time. The goal is to be able to detect when a machine acts abnormally so that further action can be taken. The algorithm used to detect anomalies should be tailored to virtual machines so that we can avoid false positives as best as possible.

The action that will be taken when an anomaly is detected should be policy-based. By that i mean that there should be parameters influencing what action is taken. These actions can range from a simple reboot of the machine to alerting an actual administrator. The goal with this policy-based decision making is not to fully replace administrators, but to try and reduce their workloads by a significant amount.

The solution should be implemented in such a way that it could work on live data. Since the goal is to use the solution to detect and solve resource hogging and other anomalies in virtual machines, it should work on live VMs and therefore free up resources live. This in turns also frees up the hands of the administrators leading to easier and simpler administrating of cloud infrastructure. Once additional resources and personnel is freed up, the overall resource utilization and system stability of the cloud should increase.

3.3 Project Scope and Phases

There are different ways to solve and reach the desired outcome described above. It is therefore important to specify what exactly will be done in this project so as to not make the scope of the project too vague or too large. Having a too vague or too large scope can lead to the project having unclear objectives where it is not entirely clear what the final outcome should look like and whether the goal of the project has been reached. It is important that the reader knows what to expect out of the project. The problem statement states that this project will be about *Designing and developing a framework which can automatically detect abnormal behavior in virtual machines, such as system faults and resource hogging, with a discussion on potential policy-based responses.* The scope is defined by the problem statement but still left quite open. The scope of the project will therefore be defined in more detail in the section below.

3.3.1 Project Scope

The framework which will be developed in this project will be designed to be able to detect anomalies in virtual machines based on their CPU usage over time. The algorithm used will be the leap detection test algorithm mentioned in the background chapter. The goal of this project is not to find the most efficient algorithm possible, but to develop a framework which works to the extent we are looking for at detecting anomalies. We are therefore not going to look at or compare different algorithms. To combat false positives, the algorithm will be changed and used in three different ways where the algorithm will have different triggers. This will allow us to do a follow-up test if the algorithm triggers to avoid some potential false positives.

The framework will be developed in Python and will be made so that it could potentially work together with a policy-based decision-making system. The main thought behind the framework is to make a semi-automated anomaly detection system and discuss how it could work together with a decision-making system to free up unreasonable amounts of resources being hogged by faulty virtual machines. This policy-based decision-making system will not be developed in this project and would potentially be the next step for further research if the prototype's performance is up to professional standards.

3.3.2 Phases Overview

The approach to solving the problem will be divided into two distinct phases. The phases will be the design phase and analysis phase. Each phase will encompass a different part and stage of the approach and together will lead to the results.

The first phase, which is the design phase will focus on five aspects crucial for the project. The first is the definition of terms which will define key terms and concepts essential for the project's understanding. This aims to help with establishing the terminology and allow for effective communication. The second aspect is framework design. The goal here it to discuss design considerations for the framework and explain how the components are going to work together. The third is to explain the importance of the prototype development and why it is a crucial part of the project. The fourth is the technology selection where the goal is to explain the selection of technologies and discuss factors leading to that choice. The last step is integration where we look at how this prototype could be integrated and how the components of the prototype work together.

The second phase of the approach is the analysis phase. This phase will explain how the model will be verified and validated to ensure that it works as intended. It also discusses the evaluation metrics which will defined the benchmarks and metrics which will be used to evaluate the framework.

3.4 Design Phase

Before diving into the intricacies of our project, it's imperative to establish a common understanding of key terms and concepts, outline the framework's design considerations, and underscore the importance of prototyping in validating our design decisions. This will be explained and discussed below.

3.4.1 Definition of Terms

In this section, key terms' meaning will be explained together with context on why those terms are important to this project. Most of the terms that will be listed here have been mentioned before, but the context of why they are used and exactly what is meant by the terms have not been explained thoroughly.

Resource hogging: This term is usually used when a program hogs or uses an increased or high amount of resources over an extended period. In this case, the one that is hogging the resources would be the virtual machine.
Resource utilization: This metric is based on the amount of time a resource, in this case a CPU, is occupied over a time interval. If a resource is occupied for 40 seconds over a one minute interval, its utilization would be 67 percent ("Resource utilization and performance", 2022).
Policy-based decision-making: The meaning of policy-based decision-making is to make a decision making process less complex and time consuming by having it follow predefined policies whenever they are applicable.
False positive: This is a classification of a result given by a test. If a result is false positive it means that a test have incorrectly stated that the condition for the test have been met. **False negative:** This is another classification of a result given by a test. This is the opposite of false positive and is when then condition for the test have been met but are not stated as such. **Memory length:** Memory length is the term defining how long the memory used in the test is. This memory represents the population used in the chi-squared test described in the background chapter. **Chi-value:** This is the value given by the chi-squared test which tells us if the value is expected or not. **Chi-limit:** This is used as a measuring point to decide if the chi-value should trigger or not. **Trigger:** A trigger appears when the test detects something that could be considered out of the ordinary based on the current parameters given at that moment.

3.4.2 Framework Design

An important part of making a framework is to first come up with a plausible design which can then be developed or implemented. This process is important as it give the designer and developer some insights into how the solution could be made and what it would require to make it. This meas that all parts of the framework needs to be thought of and planned out which in turn requires the designer to explore design considerations including architecture, components and interaction flow.

The first step to consider are the components which will be involved in the solution. To solve the problem, there are three necessary components which are required. The first one is the data on CPU utilization.

The second is the framework running the algorithm which checks the CPU utilization. The third component is the policy-based decision-making which will try to make a decision. The three components can be seen in figure 3.1.

Figure 3.1: Components

This figure shows the three main components. These are the data which is VM resource utilization, Leap detection test, and the policy-based decision-making.

The three different components require some design considerations before the prototyping can begin. The first thing which needs to be taken into consideration is what format does the VM resource utilization data come in and how can it be handled. Normally if this framework were to be put into active use, it would receive a continuous flow of close to if not real-time data on CPU utilization. This would require the framework to continuously accept new data and analyze it with the algorithm. This part is overlooked for this project as the goal is to show of a proof of concept. This is done by running the algorithm over an already existing dataset of different virtual machines' CPU usage. This means that the prototype only needs to be able to read already completed datasets. It also means that the format of the data is know beforehand and the framework can be designed to work with that specific format.

The next thing to consider is exactly how the framework and algorithm is going to function. The goal here is not to have the most efficient and accurate algorithm. Instead, this project aims to show a viable solution that works and that there is potential in this idea and type of detection. This means that although the algorithm will be tuned to show different results, its main goal is not to show the best possible result, nor work in the most efficient way. The project will also allow for the evaluation of the algorithm.

When it comes to the policy-based decision-making, consideration need to be taken regarding false negatives and false positives. It is also important to note that the aim is not to solve everything automatically, but instead making the process of responding to the algorithm less complex and time consuming by taking care of some of the decision-making for the administrator. The policy-based decision-making system will not be developed as part of the prototype but will be discussed instead.

3.4.3 Prototyping

Making a prototype is important as it is would be a preliminary version of the framework that is built to test and validate certain aspects of the design. In this case, its goal would be to validate whether the proposed solution is feasible or not. It also allows different ideas to be explored and functionality to be tested without

developing a full fledged solution. This makes it possible to visualize the proposed solution as well as to identify potential issues and in turn make informed design choices.

Different types of prototypes can be developed depending on the need. In this project, the prototype is a combination of a feasibility and a working model prototype. This means that the prototype is close to a working model but there is room for features to be added to the prototype at a later stage ("10 Types of Prototypes (With Explanations and Tips)", 2023). The reason for going this way is that while a general idea of what the prototype needs to do and how to do it is in place, there need to be room for improvement and further development of the prototype.

The prototype developed in this project will only be running the algorithm for detecting anomalies and potential crashes. The full fledged framework including the policy-based decision-making component will not be part of the prototype. It will rather serve as a test to see if this algorithm with its prototype have the potential to feed information to a policy-based decision-making tool.

3.4.4 Technology Selection

The two main technologies chosen for this project is the programming language Python and Shell scripting. Python will be the backbone of the project which the prototype will be developed in while Shell scripting will be used to used to run the prototype as well as to organize the results. Python is chosen here as it is widely used in the field and has access to a wide variate of libraries. Shell scripting is chosen due to its ease of use when reading from and writing to file. Together these two will be responsible for running the tests and making sure the outputs are organized and usable for further investigation.

3.4.5 Integration

The next step is to look at how these technologies will be integrated together. Figure 3.2 below shows an illustration of how different parts of a potential final solution would work together. The CPU data would be fed directly to the leap detection test, which in this case is the prototype. The prototype would then send information on to a policy-based decision-making tool which would make a decision based on the info it got and predefined policies. The arrow showing negative and pointing to "no action" indicates that if there are no abnormalities nothing would happen.

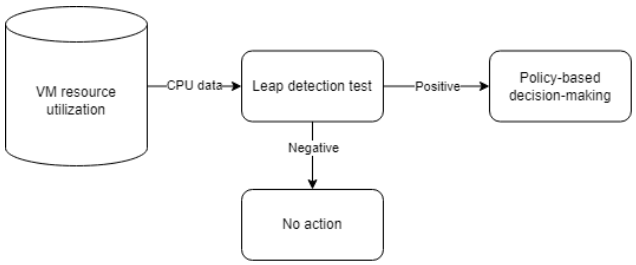

Figure 3.2: Flow

This figure shows the same three components, but with arrows indication what goes between them.

3.5 Analysis Phase

It is important to discuss how the analysis and verification of the prototype will be done and what metrics it should be evaluated by. The goal is to use the leap detection test algorithm and find out if it is suited to detect abnormal behavior and potential crashes in virtual machines. The purpose of the analysis is to analyse the results which come from the different rounds of testing that will be performed. By analysing the results, improvements can be made on the prototype before the next round of testing.

3.5.1 Evaluation Metrics and Verification

There are many metrics which can be used to evaluate the algorithm and prototype. These metrics will be used to verify if there have been improvements between iterations and to make a statement on whether the algorithm is suited for this specific task. The Rounds of Testing section in the Results chapter is where the metrics will be used. This section consists of three round of testing and improvement where the metrics evaluate whether improvement is necessary and what new discoveries have been made.

The first interesting metric which will be used is the amount of triggers per line. This gives an indication of how often the algorithm triggers and thus tells how sensitive it is. Most metrics are based around the triggers the prototype makes. Another such metric is the amount of triggers based on memory length. Evaluating this metric allows for further adjustment of the prototype. Observing the chi-value and chi-limit is also useful metrics. Much can be learned and deduced by knowing how the algorithm reacts to the CPU data and the chi-value tells that to some extent. Plots, tables and graphs will also be used during this stage. These will be used as tools to visualize and give a better understanding of how the prototype behaves.

Chapter 4

Results

In this chapter, the results of the study into using an algorithm to detect abnormal system behavior like resource hogging in a virtual machine will be presented and discussed. Data for the study was collected from 223 anonymous virtual machines, and analysis was done by utilizing the leap detection test together with shell and python scripts. The findings of this project sheds some light on what it would require to implement this solution in a real life scenario and whether this exact solution is feasible or not. This chapter begins by discussing the data which is used in the project, followed by how the prototype and testing have been implemented. It then looks at what the takeaway from the tests are and how it should be interpreted.

4.1 Data

The data is one of the most important components of this project. Without having access to actual data, experimenting with the prototype would not make much sense as the prototype requires an input of data to work. Since the importance of data has now been clarified, this section is going to look at the data used in this project in more detail. This section will look at its formatting and its size, as well as discuss a few important points about the data in question.

First and foremost, the data is a collection of CPU usage from 223 different virtual machines. The data comes in the form of a .csv file consisting of 1785039 lines or rows of data. Each row contains a timestamp which is when the data is from, followed by a value which is the CPU usage measured in CPU seconds. CPU seconds is a metric which show how many seconds the CPU was working on a process over an amount of time("How to reduce your account CPU seconds usage?", 2021). This means that if the CPU usage is 60, it spent the entirety of the last minute working. The last information on each row is the name of the virtual machine the data corresponds to. This would roughly mean that there are around 8000 lines for each virtual machine where one minute separates each new row of data. This is not entirely correct as the lifespan of the virtual machines differ where some have data for over 10000 rows and some have less. If we go off the average, each machine lives for around 5,5 days before they are deleted.

Information about the data	
Number of lines	1785039
File type	.csv
Data format	Timestamp, Value, Domain

Table 4.1: Information about the data

Shows information about the data such as the number of lines, file type and format.

One thing to point out about this data and that plays a significant role on the virtual machines lifespan is that they are part of a lab environment. This means that all the virtual machines who's data is collected in the data-sheet can be expected to behave somewhat similarly and follow certain patterns throughout their life-cycle. A normal life-cycle of a lab virtual machine would be a startup followed by installs then experiments. When those are done, the machine would be deleted and its life-cycle end. This means that the data used together with the prototype in this project will not be a correct representation of the data generated by enterprise machines.

Another important aspect of the data is that is that it is not trained or processed in any way or form. It is raw from the virtual machines running in the lab environment and have not been processed or adjusted to work or fit better for the tests this project run on the data. This means that the data will vary and not all 223 virtual machines will have usable data. This begs the question, why not use synthetic data designed to fit the algorithm? Although synthetic data will have the peaks in CPU usage we are looking for in this test and is certainly not a bad idea, it is important to start testing on real data to see how viable the tests actually are. Tests on synthetic data could could always be performed after its viability has been confirmed on real data.

This means that there is no certainty that the anomalies we are looking for are actually present in the dataset. However, it is important to note that does not make it certain that there are no anomalies present either. In essence, there might be anomalies or might not, there might be a lot in some of the virtual machines where other might have fewer.

What is important in this first round of testing using the leap detection test's algorithm is to look at how the algorithm performs when it comes to noise in the data. Noise could be referred to as the algorithm's Achilles heel. If the algorithm can not overcome noise in the data and starts giving false positives every time there is noise, its practical implication lessens substantially. This is another reason it is important to to use

the algorithm on real data first to find out whether there is potential or if the not suited for this task.

4.2 Implementation

The actual implementation of the prototype is one of the key objectives of this project. It is necessary in order for the algorithm to be tested against real CPU usage date. This would make it possible to decide whether the algorithm can be used or if it should be substituted by another. This section will focus on the implementation of said prototype and explain how it works, what it does and how the different parts of the prototype work together.

4.2.1 Technologies

The prototype is divided into two main parts. These parts are firstly the code running the leap detection algorithm which is written in Python. The second part is a shell script which is responsible for reading the data and then saving the results in a named and organized file structure, where one file is saved for the results of each virtual machine. On top of these parts, there are also python scripts which have been made for the purpose of creating graphs based on this data which can give us insight into the algorithm's performance.

4.2.2 Python script

This section is going to delve into how the python script works and how the leap detection test has been implemented into this script. It will also look at and discuss some design choices related to the script which have been made along the way. It will start of by briefly explaining what the script does then follow up whit a more detailed explanation of how it works.

To start off with, a script running the leap detection algorithm needed to be made. The original script for running the algorithm was very bare bone and contained only the key components for running the algorithm. It was therefore remade in Python to make it easier to work with and expand upon.

In its newly created state, the script consisted of the math equation for the chi-squared test, a memory length and a chi-limit. The math equation mentioned is as following:

$$\chi = \sqrt{\frac{(\text{sum} - \text{Memory\_Length} \times \text{diff})^2}{\text{Memory\_length} \times (\text{Memory\_Length} + 1) \times \text{avg}}}$$

These three components are what makes the basis of the leap detection algorithm. The math equation is what calculates the chi-value which is matched up against a chi-limit. If the value is higher than the limit, a trigger is made. This test compares a new value to known values, hence a memory is needed. In this first iteration of the prototype, the chi-value was calculated with a memory length of 10 and the chi-limit was set manually. Now that the basis and starting point of the algorithm and prototype have been briefly explained,

it is time to move on to the prototype's development and its evolution over the different iterations. The first step in this process is to explain how the prototype works and how the algorithm was implemented in the prototype.

The prototype consists of five main parts. These parts are initialization, command-line argument parsing, functions, main loop, and final output. The first part which is initialization does exactly what its name implies. This is the top of the script where variables are initialized and arrays are created. Following the initialization part comes command-line argument parsing. Command-line arguments are used to to run the prototype with different settings and to manually set some of the variables such as chi-limit and memory length down the line. The third part is the functions. These functions are responsible for the prototype's output as well as different run modes which will be introduced shortly. The fourth part is the main loop of the prototype which runs the actual testing. This loop is responsible for calculating the chi-values, handling the memory, calling the correct functions and reading the input data. The final part named final output is just responsible for printing the information which have been gathered throughout the current test.

Since the different parts of the prototype have now been explained briefly, the next step is to look at how it actually operates and how it reached its first test ready iteration. For starters, the main loop is what does the calculations and handles the incoming data. This is a for-loop which loops over all lines of incoming data. It reads a new line of data for each round in the loop until there are no more lines. At that point the loop ends and the gathered data is printed. As mentioned, this is where the data is read and handled. The first step the prototype takes here is handling the data. That means separating the CPU value from the rest of the data and setting the current CPU value as a starting point. Since the data is presented cumulatively, the prototype always subtracts the previous CPU value from the current before it proceeds with the calculations. This step is crucial since the algorithm is interested in current values and not the sum of all previous values.

Once the data has been handled, the loop continues on to handling the variables used in the algorithm such as calculating the sum of all CPU values stored in the memory, as well as calculating the average value of those stored in memory. It then finds the chi-limit and reaches the part of the code where the chi-value is calculated. In figure 4.1 below, the code which calculates the chi-value and then handles the memory is displayed. This part of the code is only reached if the memory is already full. This is because the sum and avg variables would not be correct if that weren't the the case. Once the chi-value is calculated, the oldest entry in the memory is replaced with the newest value. The same goes for the sum and avg variables.

```
chi = math.sqrt((sum - MEMORY_LENGTH * diff) ** 2 / (MEMORY_LENGTH *
(MEMORY_LENGTH + 1) * avg))
index = index % MEMORY_LENGTH
sum = sum - memory_buff[index] + diff
avg = avg - (memory_buff[index] / (MEMORY_LENGTH + 1))
memory_buff[index] = diff
```

Figure 4.1: Code for calculating the chi-value and updating the memory and variables

This figure shows the code calculating the chi-value, then handling the memory, and sum and avg variables.

To prepare for the first tests to be done, two modes were implemented inside this loop before it reaches the chi-value calculation. These two modes were a max function and a min function meant to replace the manually set chi-limit. The goal of these two functions were to set a chi-limit based on what was recorded in the memory. Setting a chi-limit manually for each test and VM was impractical so these functions were implemented so that the prototype could set the limit by itself. Figure **??** below shows how this is done. The two modes are called extra mode 1 and 2 as they are an addition to the prototype. They are initiated by using the command-line arguments *-min* and *-max*. They work by respectively calling a $find_min()or$

Before moving on to another part of the script, there was one additional change which was made for the calculation of the chi-limit. This was as a response to the still low chi-limit and consisted of observing the chi-values of the virtual machine before setting a static chi-limit. This was done by remembering what the highest chi-value observed was and then after having observed for a while, setting the chi-limit just beneath that value. This was done as shown in figure 4.2 by running the normal max function for the first part of the test and then switching over to the highest chi-value observed rounded down to the nearest whole number.

```
if counter < 7500:
    mem_max = find_max()
    max_avg = avg + mem_max / (len(memory_buff) + 1)
    tchi = math.sqrt((sum - MEMORY_LENGTH * mem_max) ** 2 / (MEMORY_LENGTH *
    (MEMORY_LENGTH + 1) * max_avg))
    CHI_LIMIT = tchi * 2

elif counter == 7500:
    rounded_highest_chi = math.floor(highest_chi)
    CHI_LIMIT = rounded_highest_chi
```

Figure 4.2: Code for setting the static chi-limit

This figure shows the code for setting the static chi-limit after an observation period.

After the chi-limit and chi-value have been calculated, the loop gives a call to the primary function in the functions part. This is where the comparison of the chi-limit and chi-value are done. It is where the outputs are decided on and made. This part consists of four main parts which are called verbose, quiet, all events and plotter. These different parts are called by using command-line arguments when the prototype is executed. All the different modes compares the chi-value and chi-limit but only one them is to be used at any given time. If the quiet mode is enabled, the output form the prototype will be a list of ones where each one represents a trigger. The all events mode will list all lines with ones and zeros where the ones represent lines with triggers and zeros lines without. The verbose mode is the one that gives the most information. This mode prints information about each line of data, what numbers are stored in memory and the chi-limit and chi-value. It also informs on how many lines were read and what number of those included triggers, as well as what the highest chi-value was. The last mode which is the plotter mode is designed to make an output which can be used to make plots and graphs of the information later. This includes the date and time of data as well as the CPU values and when it triggers. This information is then organised so that it is easy for a plotter script to read.

4.2.3 Part 2 - Shell script

To make testing and organizing easier, shell scripting was used to actually run the prototype. A few shell scripts with small differences were made to organize the outputs from the prototype as well as to make sure the data it received was formatted correctly. Since the data used for this project came as one single .csv file, it was first handle in a script to make it easier and because the prototype would not operate of .csv files in a professional setting. The script separated the data using AWK and CUT then fed it to the prototype. In figure 4.3 below, the main mechanism of the shell script used to run the prototype can be seen. What it does is to create a folder named after the date and time the script was ran as well as what command-line arguments were used. It then makes sure that the outputs it receives form the prototype are separated in to unique files named after the virtual machines. This is done to easier sort and investigate the results later. This altogether made running the prototype multiple times easier.

```
current_date=$(date +%Y-%m-%d-%H-%M-%S)
mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag"
cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag"
|| exit
for ((i=0; i<=223; i++))
do
    formatted_number=$(printf "%04d" $i)
    output_file="VM-$formatted_number.txt"
    echo "$formatted_number"

    awk -F ',' '$3 ~ /VM-'"$formatted_number"'/' ../result-timestamp-changed.csv
    | cut -d ',' -f 2,1 | python3 ../main.py $max_flag $min_flag $avg_flag
    $verbose_flag $plotter_flag > "$output_file"

done
```

Figure 4.3: Shell script for running the prototype

This figure shows the shell script used to run the first test iteration of the prototype.

As mentioned, there were additional shell scripts with small changes from the original script showcased above. The first of these scripts has its changes depicted in figure 4.4 below. This shell script has been changed to only run the test on one virtual machine, but to do it with 26 times with a memory length ranging from 5 to 30. This was done to get an understanding of how different memory lengths impacts the algorithm. The last version of the script is shown in figure 4.5. This version is changed to only run once and only read the values of one virtual machine. This was done to do further investigation into one machine with the last iteration of the prototype.

```
    current_date=$(date +%Y-%m-%d-%H-%M-%S)
    mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag
----+_memory_test"
    cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag
----+_memory_test" ||       exit
    for ((i=5; i<=30; i++))
    do
    formatted_number=$(printf "%s" "$i")
    output_file="VM-0008_mem_$formatted_number.txt"
    echo "$formatted_number"
    awk -F ',' '$3~~/VM-0008/' ../result-timestamp-changed.csv | cut -d ','
    -f 2,1 | python3 ../main.py $max_flag $min_flag $avg_flag $verbose_flag
    $plotter_flag -m $formatted_number > "$output_file"
done
```

Figure 4.4: Shell script for running memory test

This figure shows the script that runs the same test starting with a memory length of 5 and ending with a memory length of 30.

```
    current_date=$(date +%Y-%m-%d-%H-%M-%S)
    mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag
----+_adaptive"
    cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag
----+_adaptive" || exit
    formatted_number=$(printf "%s" "30")
    output_file="VM-0008_adaptive.txt"
    echo "$formatted_number"

    awk -F ',' '$3~~/VM-0008/' ../vm-0008-orig-csv.csv | cut -d ','
    -f 2,1 | python3 ../adaptive_main.py $max_flag $min_flag $avg_flag
    $verbose_flag $plotter_flag -m 30 > "$output_file"
```

Figure 4.5: Shell script for running the latest iteration of the prototype

This figure shows the shell script used to run the last iteration of the prototype.

4.3 Rounds of testing

This section will be dedicated to the results gained from running the prototype on the data. It will delve into these results to examine and study them. The section will be divided into multiple parts where each part will represent one instance of settings in the prototype. The settings which each instance will represent are the chi-limit and its calculation, as well as the memory length used in the prototype. As explained in the section about the prototype, the chi limit determines when the algorithm triggers while the length of the memory is used to adjust how sensitive it will be based on previous values

Once the settings of an instance have been explained and discussed, the rest of the individual instance will be comprised of the actual results of that run of the prototype. This will include graphs which showcase the frequency and distribution of triggers based on amount of triggers per line, as well as graphs that show triggers over time in correlation with CPU value and chi-limit. This data will provide plenty of information which then will be used to discuss the results and the algorithms application in the field. This information will also be used to fine-tune the chi limit and memory length for the next round. This process will continue for a couple of rounds and will be summed up in a final discussion. The final discussion will delve into whether the algorithm is suitable for the task.

4.3.1 Round 1

The results of the first test will be the least refined results and will simply function as a way to find out whether it is feasible and to get an understanding of what needs to be done next. As is common in prototyping, there needs to be multiple iterations. In this case, the prototype is done and the iterations now refers to changes in in the prototype which affects mostly the chi-limit and memory length. This is what is meant by the different rounds explained earlier. In this first round, the chi-limit and memory length is set to a value with little fine-tuning done beforehand. This allows us to see what values might create problems and what need to changed before the next round.

Before actually investigating and discussing the results of the first round, it is important to know what values have been used for the chi-limit and memory length for this round of testing. How the chi-limit gets calculated have been explained in the prototype section and for this test the same calculations are done. The only addition to the regular equation for the chi value is that the limit is multiplied by 2. This means that the chi-limit is now the following equation.

$$\chi = 2 \times \sqrt{\frac{(\text{sum} - \text{Memory\_Length} \times \text{diff})^2}{\text{Memory\_length} \times (\text{Memory\_Length} + 1) \times \text{avg}}}$$

The reason for this is that the chi-limit by itself is too low and would cause the algorithm to trigger on close to every line of data. Multiplying it by 2 reduces the amount of triggers by a huge margin and gives results which can better be analyzed. The memory length is set to 10. Having a longer memory length would lead

to the algorithm taking longer to adjust to lasting changes in the data, while a short length would make it adjust faster. This simply means that if the memory length is longer, the algorithm will trigger multiple times before adjusting to lasting changes in the data. And example of this would be if there was a sudden rise and then a plateau in the data. A long memory would continue triggering until the new higher values have become the norm, while a short memory would only trigger once to a couple of times and then forget that the previous values were much lower. Using 10 here functions as a base value which can be adjusted up or down in later rounds of testing.

The first results to look at is show in figure 4.6 below. This figure shows the results of running the algorithm on one of the virtual machines using the function which bases the chi-value on the highest value in memory, the *-max* tag. Using the *-max* tag when running the code will on average give a higher chi-limit than the *-min* tag. This in turn should give a lower amount of triggers from the algorithm.

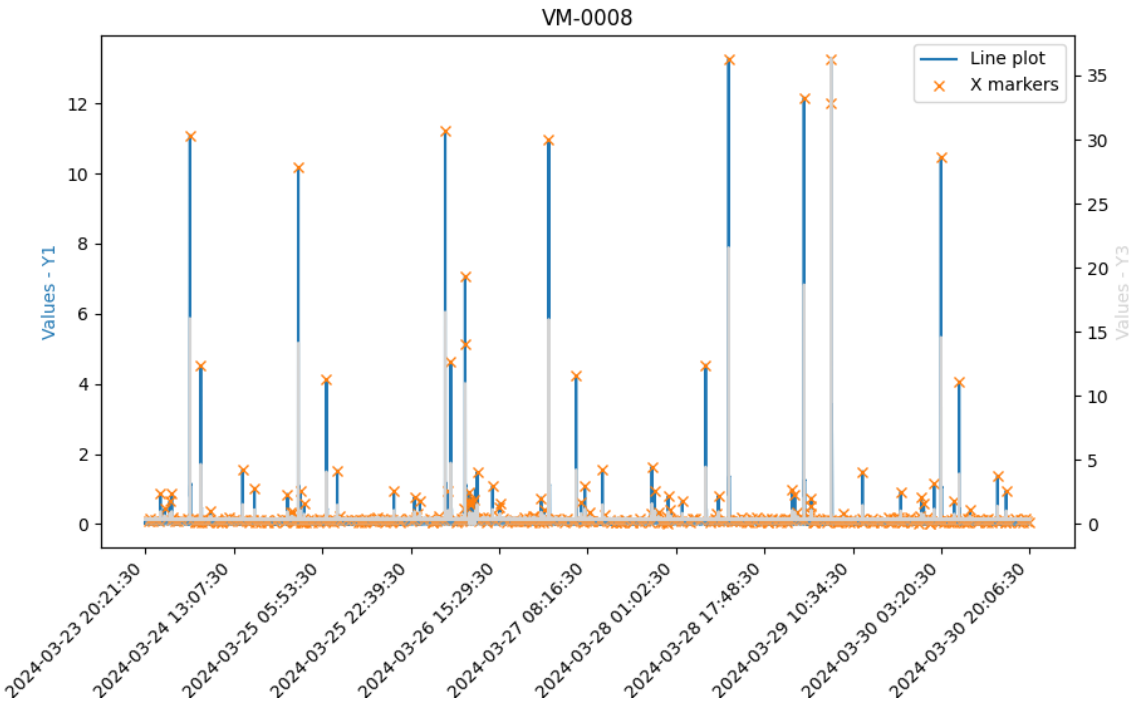

Figure 4.6: Triggers form VM-0008

This figure shows the triggers from the algorithm marked with orange X's while the blue line shows the chi value and the gray area which looks like a gray line shows the CPU value.

After having looked at the figure, it is clear that there is a huge amount of triggers and that they appear consistently though the whole lifespan of the virtual machine. One important thing to note is that the time span the graph in the figure covers is seven days. This means that although it looks like it triggers all the time, this is not true. To get a better understanding of how often it actually triggers, it is important to look at the triggers over a shorter amount of time. The figure 4.7 below shows the same data as the previous

figure, but has been changed to only show the first quarter of the data. This means that the information now spans over roughly two days instead of one week. This allows for a more accurate and detailed graph because it is less compacted, which gives the benefit of being able to distinguish triggers from one another together with more accurate time-stamps.

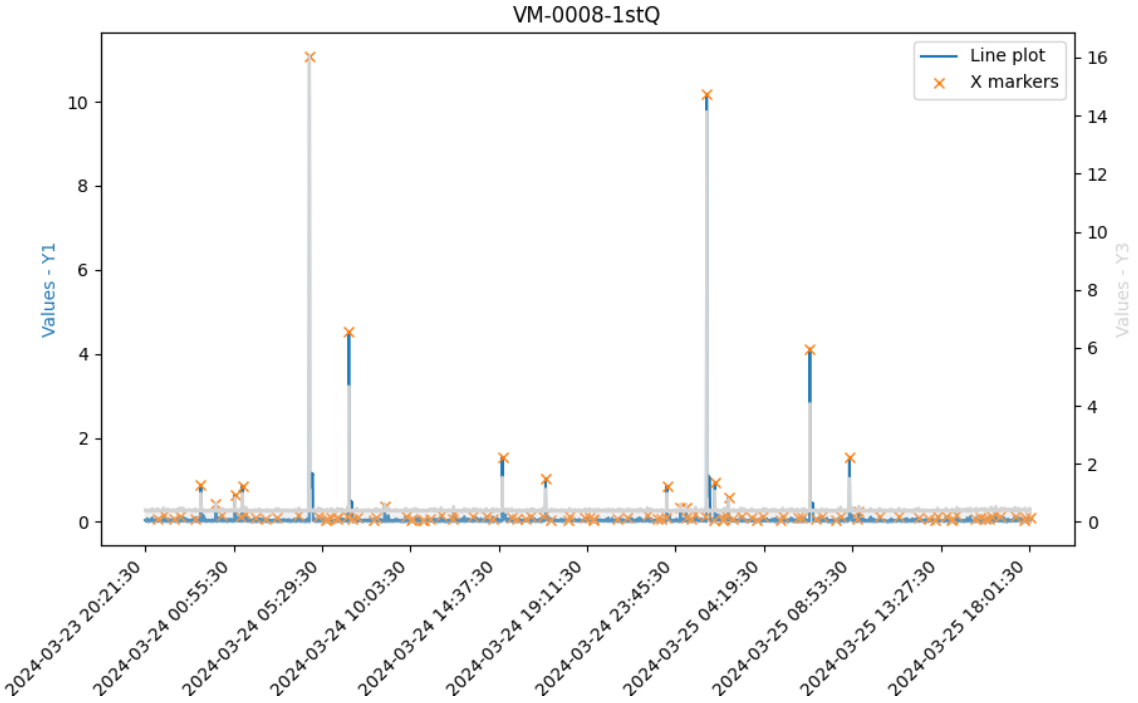

Figure 4.7: Triggers form VM-0008 1st. quarter

This figure shows the triggers from the algorithm marked with orange X's while the blue line shows the chi value and the gray area which looks like a gray line shows the CPU value. It is the first quarter of the previous figure.

As mentioned, it is now possible to distinguish a lot of the different triggers from each other, but there are still a lot of triggers. Most of the triggers appear with little to no visible changes in the CPU value which is the gray line in the graph. Although there are some triggers that appear on top of CPU spikes, there still seems to be a larger amount of triggers than the desired outcome. The desired outcome we are looking for is one where it only triggers when the CPU acts abnormal. This means that although it triggers on the spikes, the algorithm is still too sensitive.

This statement is further enhanced if we look at the the figure 4.8 below. This plot show the number of triggers per line on the x-axis and the frequency of how many virtual machines this ratio appears in on the y-axis. The red graph in the distribution plot symbolises the max function while the blue graph symbolises the min function.

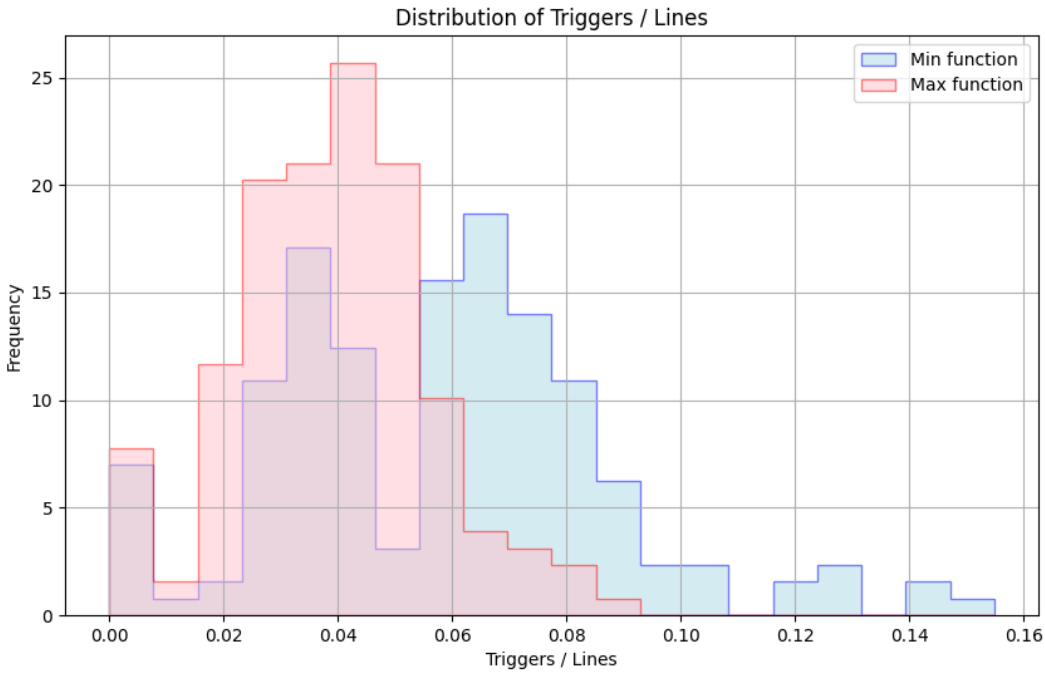

Figure 4.8: Amount of triggers divided by line for all VMs with *-min* and *-max* tag

This figure shows the amount of trigger divided by line for all VMs with the *-min* and *-max* tag The red represents the max function and blue the min function.

After investigating this plot, it is clear that the min and max function gives different results. A lot of knowledge can be gathered from this plot which can then be used when tuning the script for the next round. It appears that according to the plot, the *-max* function has a lot lower spread than the *-min* function. There are a couple of reasons and aspect of the data which might attribute to this being the case. The min function might have a low enough threshold to trigger on additional virtual machines which the max function ignores. The graphs also tells us that the max function triggers on average fewer times per line than the min function. This means that the concept of the max and min function works as intended.

Even though they work as intended, we can see that they trigger quite often which, can be said to be too often. Once the chi-limit has been changed, the amount of triggers per line should go down significantly enough to give the graphs a visual difference in the next round of testing.

Now as mentioned before, we can see that the chi-limit needs to be changed for the algorithm to get closer to what is needed of it. The limit needs to be higher than it is right now. The length of the memory might also need to be tampered with as that affects how many triggers appear when the CPU usage increases for more than one minute. The trigger for this test was 10 which means that it is somewhat sensitive to changes. Since it looks like there are few actual spikes in CPU usage, extending the memory might also help in lowering the

amount of triggers. This is because extending the memory would lead to each specific value having less of an impact on what the chi-value and chi-limit would be.

Another point that should be made clear here is that since the CPU value is measured as CPU seconds per 60 seconds, it might not be the most optimal to make a program that only triggers when it reaches 100% or 60 which is the equivalent. The best scenario then is to look at its previous usage and then take its current value and compare it. This allows for a dynamic detection and triggering which is not hindered by having to reach a threshold.

4.3.2 Round 2

In this second round of testing, experimenting with the prototype and extra analysis of the data have been done. This has been done to get a better understanding of the prototype and data based on the results that came out of the first round of testing. The first round of testing left a lot to be desired considering the algorithms sensitivity. The first round showed that it does indeed trigger when the value changes, but it triggers too often. It needs to be adjusted so that it triggers only when the changes in CPU seconds is large enough to actually matter.

There are two main factors that decide when the prototype triggers, memory length and the chi-limit. This means that improvements can be made to the prototype by changing the memory length and by adjusting or changing how the chi limit is calculated. In the first round the chi-limit was based on the min and max functions. These functions bases the chi-limit on what the highest and lowest values stored in the memory are. This means that the chi-limit these functions make would be influenced by the memory's length. One way to progress with this would be to experiment with different memory lengths and then take a new look at whether the chi limit need to be adjusted further.

The two figures below, namely figure 4.9 and figure4.10 are results of changing the memory length. To do this more in-dept testing, it was decided to investigate further using just one virtual machine instead of doing all the testing on all the virtual machines in the dataset. This decision was made to be able to optimize the prototype faster and because the same points that are found from this one VM will apply to all the virtual machines. The virtual machine in question here is VM-0008 which is the same VM as the one discussed in detail in round 1 earlier. Also since the goal here is to optimize the prototype with the algorithm and achieve fewer triggers as of now, it has been decided to use the only the max function in this second round of testing as that had the least amount of triggers.

If we look at 4.9 we can see that the plot progresses from 5 to 30 on the x-axis which is the range of memory lengths used in this experiment. The y-axis here represents the number of triggers which appeared for each memory length. Another piece of valuable information before looking further at the graph is that the total number of lines in the VM is 10075. This means that when the memory length is only 5, the prototype

triggers on about 12% of the total amount of lines with 1212 triggers as seen in table **??**. This improves drastically at the start as the memory length grows before slowing down when the length reaches about 20. Going by how this graph develops, we can deduce that if the memory length increases even further beyond 30, the number of triggers will stabilize at a number slightly lower than it currently reached with a memory length at 30. When the memory length is set to 30 and the max function is used, the number of triggers made by the prototype is 117. That means that it has gone from triggering on roughly 12% of the lines to close to 1%.

If we look at the second graph below shown in figure 4.10 we can see a graph with the same x-axis which comes from the same test of different memory lengths. This graph however shows the highest recorded chi-value for each run with different memory length. It shows that the highest ch- value recorded when the memory length is 5 is 9.79 and that it is 22.39 when the memory length is 30. The difference in highest chi-value between the different memory lengths is a direct result of the memory length and the reason why the number of triggers is so different between them. Increasing the memory length would make the highest chi value even higher and at the same time decrease the number of triggers even further.

This is because when the memory length is increased, more values are added to the population. These values which are now stored in greater numbers are the values which the max function which was used in this test looks at when setting a chi-limit. It chooses the highest value stored in memory to set the chi limit, where as the function which calculates the current chi-value uses the value of the current line which would represent the current CPU value in a real scenario. This means that each time the memory gets longer, the change for a trigger gets lower.

Another point to take notice of is the steep increase in the highest chi-value recorded. As seen in figure 4.10 and table **??**, the highest chi-value increases with memory length as stated earlier. This is because of the way the chi-value is calculated which is displayed here:

$$\chi = \sqrt{\frac{(\text{sum} - \text{Memory\_Length} \times \text{diff})^2}{\text{Memory\_length} \times (\text{Memory\_Length} + 1) \times \text{avg}}}$$

Each time the memory length gets increased, the sum variable which is the sum of all values stored in memory will also increase. This increase in the sum is what drives the increase of the chi-value, as its increase will almost always be lager than zero and can be as high as 60. This means that generally as the memory length increases, so will the maximum chi-value.

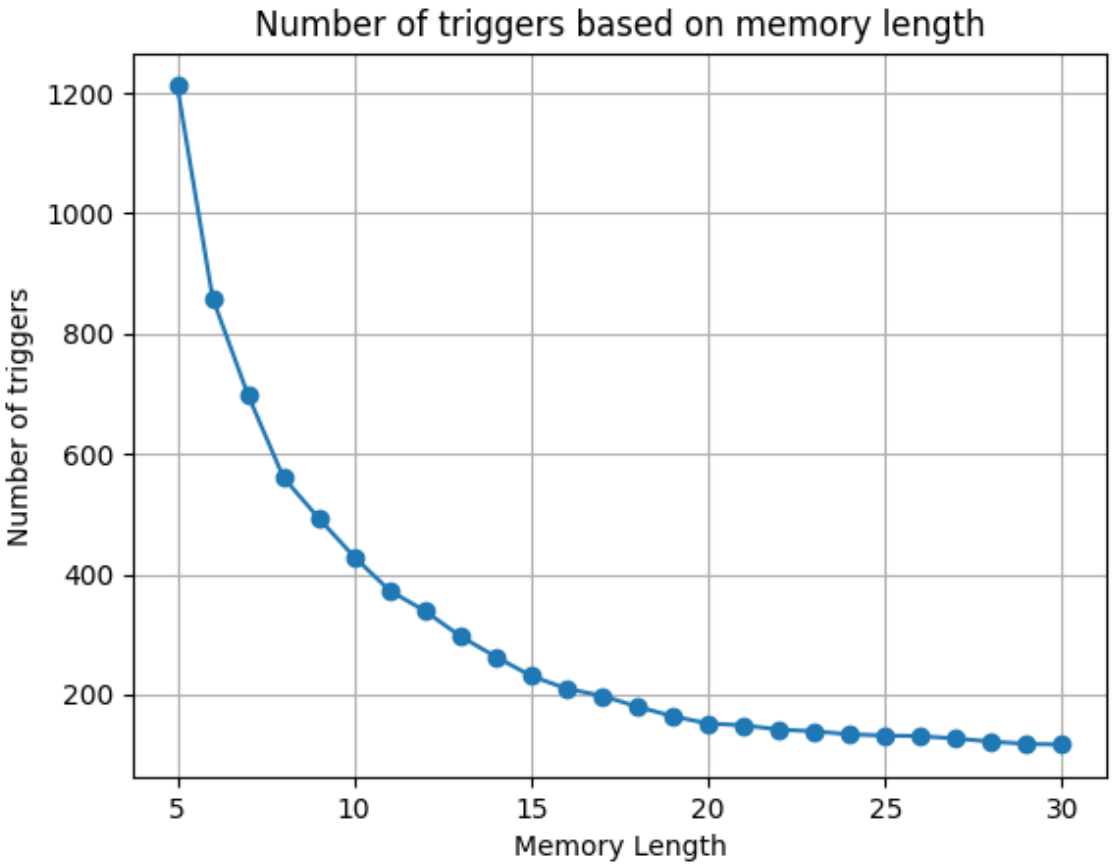

Figure 4.9: Number of triggers based on memory length

This figure shows the number of triggers based on memory length ranging from a length of 5 to 30.

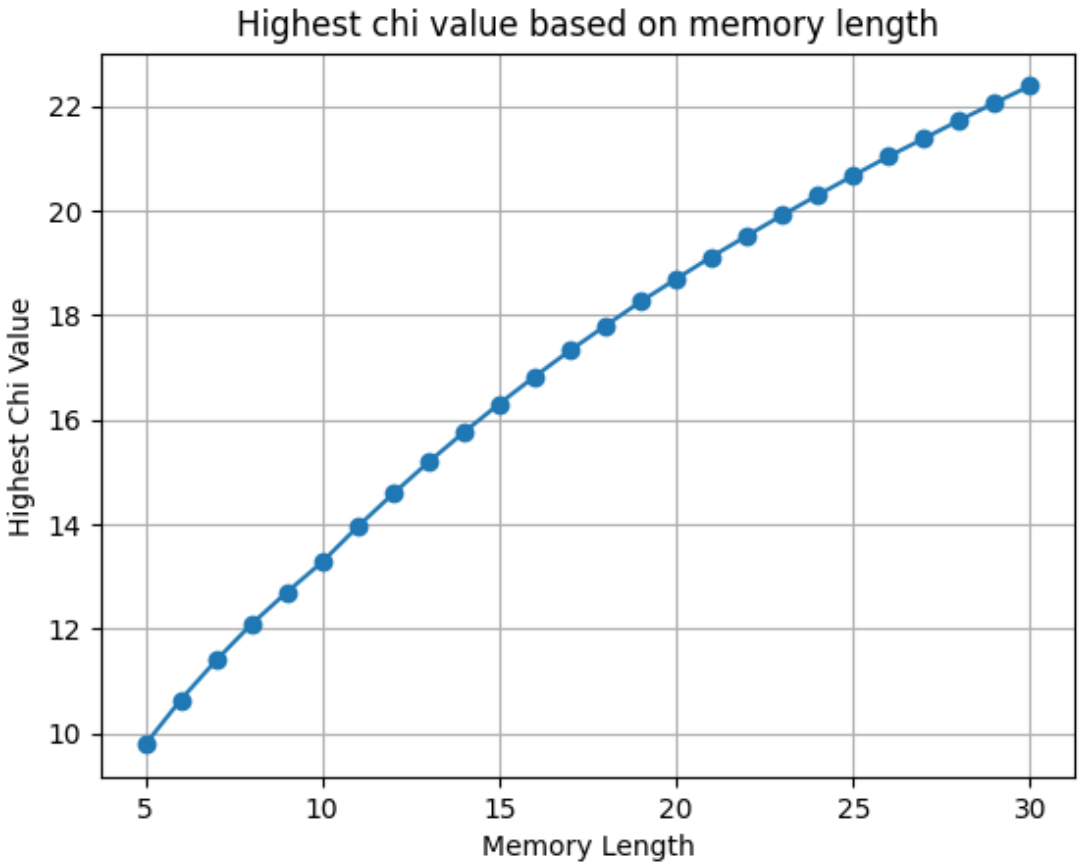

Figure 4.10: Highest chi value based on memory length

This figure shows the highest recorded chi-value based on memory length ranging from a length of 5 to 30.

Table 4.2: Memory lengths effect on the prototype

Memory lengths effect on the prototype		
Memory length	**Number of triggers**	**Highest chi value**
5	1212	9,786
6	859	10,640
7	698	11,396
8	561	12,088
9	493	12,701
10	429	13,278
11	373	13,953
12	339	14,583
13	297	15,187
14	263	15,758

Memory lengths effect on the prototype (Continued)		
Memory length	**Number of triggers**	**Highest chi value**
15	231	16,304
16	210	16,825
17	198	17,323
18	180	17,798
19	164	18,261
20	152	18,692
21	149	19,118
22	142	19,519
23	139	19,914
24	134	20,297
25	132	20,669
26	131	21,037
27	127	21,380
28	122	21,727
29	118	22,061
30	117	22,386

There are a lot of interesting points and useful information which have been discovered so far in this second round of testing. It is clear that increasing the memory length helps reduce the amount of triggers, at least while using adaptive functions to set the chi-limit. This could be improved further by adjusting the memory length even further and by artificially changing the chi-limit by multiplying or dividing this expression further:

$$\chi = 2 \times \sqrt{\frac{(\text{sum} - \text{Memory\_Length} \times \text{diff})^2}{\text{Memory\_length} \times (\text{Memory\_Length} + 1) \times \text{avg}}}$$

In both rounds of testing so far, this expression have already been multiplied by two to keep the number of triggers down. This means that it is possible to influence the triggers by increasing or decreasing this value.

The graph in figure 4.7 below shows the the amount of triggers for VM-0008 when using the max function and having a memory length of 30. We can see that the amount of triggers are drastically reduced from the first round of testing which have already been discussed above. There are however, still quite a lot of triggers which are caused by small changes in the CPU value. This means that there is still room for improvement.

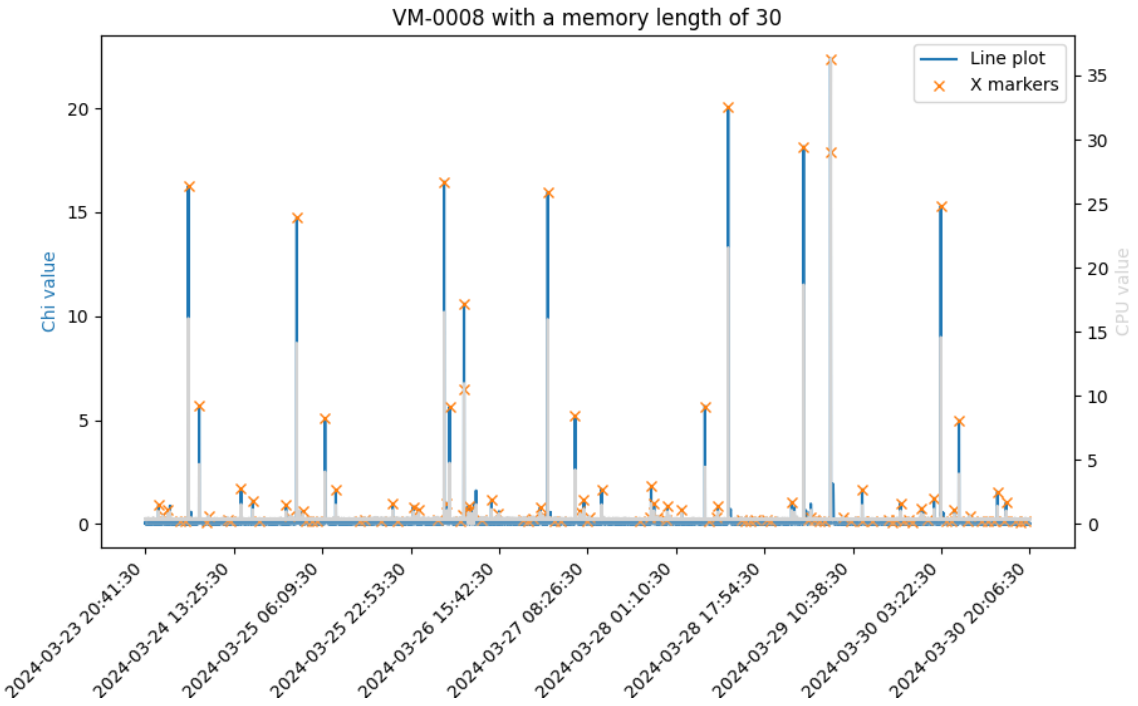

Figure 4.11: Triggers from VM-0008 with a memory length of 30

This figure show triggers from VM-0008 with a memory length of 30

One last test was done during this second phase and that was to see how the different memory lengths would perform if the chi-limit was set to a fixed number instead of relying on the min or max function. The chi-limit used in this test was the highest chi-value from the test with 10 as the memory length rounded down to the nearest whole number. The chi-limit was therefore set to 13 which instantly meant that there were zero triggers for all tries with a memory length shorter than 10 as those did not get any chi-values at 13 or higher.

Figure 4.12 below shows what the graph looks like when the memory length is set to 30 and the chi-limit to 13. This graph contains 11 triggers which is a huge improvement over the previous test where the chi-limit was defined by the max function. The problem with this approach is that although it get good results, different virtual machines will have different highest chi-limits. This means that this approach only works when it is used on one virtual machine where some test have already been done to determine what a good threshold would be.

Figure 4.12: Triggers from VM-0008 with a memory length of 30 and chi limit set to 13

This figure show triggers from VM-0008 with a memory length of 30 and chi-limit set to 13

This round 2 gives a good amount of information on what can be done to improve the prototype for the third round of testing. Using a longer memory length appears to generally be a good idea. It would also be beneficial to make the prototype adapt itself to fit the virtual machine it is watching. This could be done by having the prototype set its own stationary chi-limit after having observed the virtual machine for a while. It could still be allowed to trigger normally during the observation period, but those triggers could be considered less significant than those who appear after the observation period.

4.3.3 Round 3

In this third and last round of testing, the points discussed in round 2 are implemented. This includes having the prototype set the chi-limit based on an observation period and using a memory length of 30. Testing is also done with modified data which aims to replicate a virtual machine which get stuck at 100% CPU utilization or in this case a CPU value of 60 which is the equivalent. The goal with this last round of testing it to get an idea of whether this algorithm and approach is suitable and could become applicable in the professional field.

Below are two figures, namely figure 4.13 and figure 4.14. Both these figures are test results of using the adaptive prototype which sets a consistent chi-limit after having observed the virtual machine for 7500 lines. The memory length is also set to 30 in both examples. The same virtual machine was used in this test as

in the previous, namely VM-0008. Figure 4.13 is of the adaptive prototype on the original VM-0008 while figure 4.14 is the same adaptive prototype but with a modified VM-0008. This modified VM-0008 has been changed to have maximum CPU usage on all lines from line 8000 until the end. This is as mentioned to replicate a virtual machine which crashes and gets stuck at maximum CPU usage.

We can see on the graph in both figures that the amount of triggers gets reduced drastically on the last 25% of the lines. The drastic reduction in triggers is because the chi-limit which is set after 7500 lines is the highest chi-value recorded up till that point rounded down to the nearest whole number. In the case of this VM, the chi-limit is set to 20 as the highest chi-value recorded was 20.049. In figure 4.13 which is the original VM-0008, only one trigger is recorded after this limit is set. That trigger is the last marker visible in the graph and had a chi limit of 22.386. This trigger appeared after having recorded a CPU value of circa 33 two minutes in a row. Since the CPU value is in CPU seconds per 60 seconds, that means that the virtual machine had the CPU running on full power for half a minute two minutes in a row. Now this does not necessarily mean that there should be a trigger there, but the algorithm and prototype in its current form places a trigger there. After this trigger, no new triggers appear.

When it comes to the second graph in the figure 4.14, it looks a little different. The data or virtual machine used in this test is the same VM as in the previous, but it has as mentioned been modified. As seen by the gray area in the graph, the data has been modified to have a maximum CPU usage after line 8000. The gray area gets therefore stuck at 60 for the rest of the graph which indicates that the CPU was running at maximum load the entire time. This test has four triggers after the observation period has ended and the chi-limit has been set. The first trigger is the same as the on discussed previously which appears in figure 4.13 after the observation period. The other three triggers appears at the three first lines where the CPU value is set to 60. The highest chi-value comes the first time 60 appears and then it gets lower for each following line. After three lines, there are enough high values in memory for it not to cause a trigger.

Figure 4.13: Triggers from the original VM-0008 with a memory length of 30 and adaptive chi limit

This figure shows the triggers from the original VM-0008 with a memory length of 30 and adaptive chi-limit

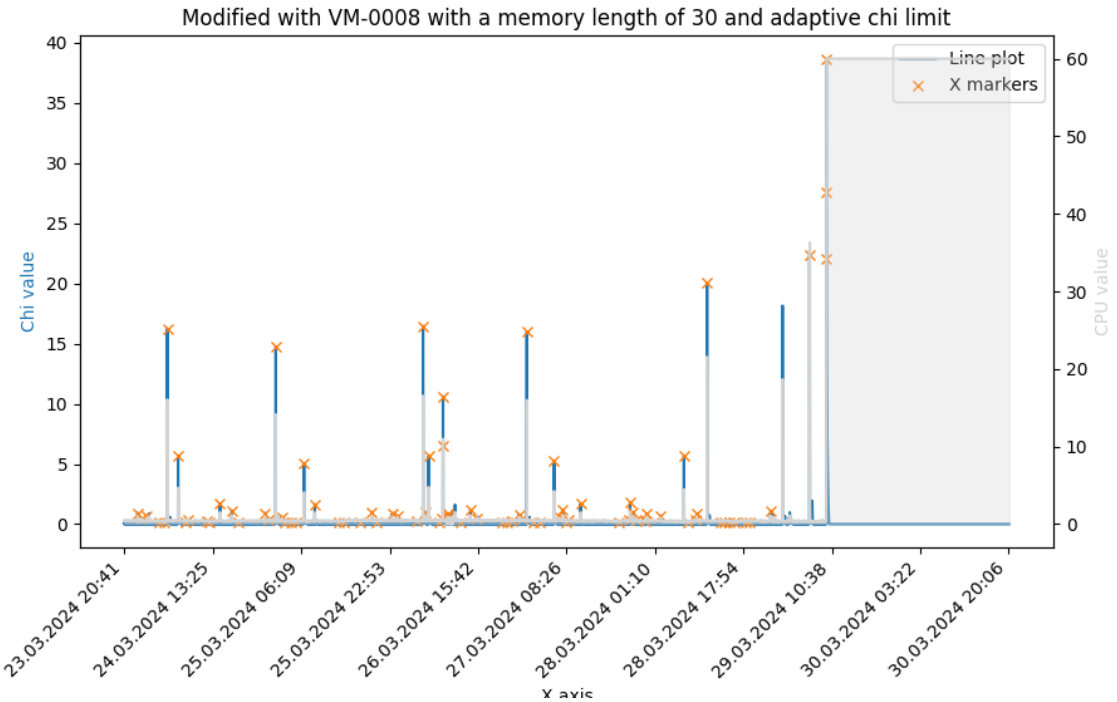

Figure 4.14: Triggers from modified VM-0008 with a memory length of 30 and adaptive chi limit

This figure shows the triggers from modified VM-0008 with a memory length of 30 and adaptive chi-limit

In this third round of testing, lessons learned from the two previous rounds of testing were Incorporated into the prototype and were reflected in the results. If we ignore the triggers during the first part of the tests which is the observation phase, the prototype shows clear signs of improvement. Triggers which were happening on low CPU values stopped appearing. It also acted as expected while reaching the plateau in CPU values by triggering a couple of times before adjusting to it. All this can be tweaked even further by adjusting what chi-limit is set to after the observation phase and by tinkering with the memory length even further. Increasing the memory length would make it trigger for a longer period when reaching the plateau before having adjusted to the increased CPU values, whereas shorter memory would make less triggers appear.

Changing the memory length here both ways could have positive and negative effects. Increased memory length here could lead the one trigger which appear after the observation period to trigger twice instead of once. The moment this trigger appears, the CPU value is increased for a couple of minutes, which means that a longer memory would trigger a few more times before having adjusted to the increase in CPU value. A shorter memory would probably only trigger once here but would instead trigger more often in total. That is because it would be more affected by each line because it remembers fewer line. A small jump in CPU usage would therefore have a larger effect if it comes after a period of close to zero CPU usage since the prototype have forgotten that there used to be higher CPU usage.

4.4 Takeaway

During these three rounds of testing the algorithm has been tested in different ways through the use of the prototype. The mathematical expression to calculate the chi-value has stayed the same while the different parameters which affects this expression have changed. Different approaches and scenarios have been explored with the prototype going through different iterations. There is a lot of information which can be taken away from these three rounds of testing.

In the first round of testing we looked at how the algorithm performed while using the max function on the virtual machine VM-0008 with a memory length of 10. It was made clear during this test that further investigation was necessary due to the high number of triggers. It was hard to distinguish triggers from one another, even while only looking at one quarter of the period which the virtual machine operated in. We also looked at how frequent all the virtual machine in the dataset triggered while using both the min and max function. The max function seemed to perform better here so it was decided to continue testing while using that function. Since the amount of data was so large, it was also decided that the next rounds of testing would focus on improving the prototype against one specific virtual machine, namely VM-0008.

The focus in the second round of testing was to find out what impact memory length had for the algorithm. It was therefore experimented with different memory lengths ranging from 5 to 30. The original memory length used was 10 so this test started at 5 lower than the original and ended with a length which was three times longer. This test showed that initially an increase in memory length would drastically reduce the amount of triggers, but as the memory got longer the benefit from increasing it grew smaller and smaller. At the beginning a difference with one in memory length meant 353 less triggers while at the end that same difference in memory length amounted to one less trigger. This meant that increasing the memory length at this point would have little effect on the overall amount of triggers.

Another point that was investigated during this second round was the chi-value. Since there still were a sizable amount of triggers, investigating the chi-value would be beneficial before further tuning the chi-limit. This revealed that while the memory length increased so did the chi-value, and in turn chi-limit since that was calculated similarly to the chi-value. The highest chi-value increased by over 10 from the shortest memory length to the longest. This information and the fact that different virtual machines would get different highest chi-values meant that having a predefined chi-limit would not work.

The same graph that was plotted during the first round of testing was also plotted here using a memory length of 30, and it showed significant improvement its predecessor. Although it was an improvement, there were still too many triggers on low CPU values so it was tested with a static chi-limit which turned out to perform much better. This meant that using a memory length of 30 and setting a static chi-limit for each virtual machine would be optimal.

In the third round, the prototype had been modified to set a static chi-limit after having run with the max function first. This chi-limit was set after 7500 lines or 75% of the data for this VM. The period before the limit was set was be the observation period where the prototype observed the virtual machine and remembered the highest chi-value observed. After 7500 lines, this value was then rounded down to the nearest whole number and set as the static chi-limit. The prototype was allowed to trigger normally during the observation period before switching over to the static chi-limit after. One test was done with the same virtual machine as earlier and it showed that after the observation period, only one trigger appeared. That trigger was also somewhat significant as it appeared when the virtual machine experienced semi-heavy load for over two minutes.

The last part of the third round was about testing how the prototype would react to data that had been modified to represent a plateau in which the virtual machine would get stuck at maximum CPU usage for an extended period of time. In this test, the plateau would appear a little bit after the observation period had ended. The prototype triggered three time in rapid succession when it reached the plateau before going quiet.

During this three step process the prototype underwent different changes and ended up at a place where it could potentially be useful. Although there is still room for improvement and fine-tuning which would require furthered experimenting with the prototype and algorithm, it has reached a point where it would potentially be useful at detecting short spikes in CPU usage. Unfortunately that is also all it seems to be useful for. The problem here is that it triggers the moment a spike appears and only when spikes appears. This means that it is not capable of signaling a potential crash expect for at the exact moment it happens. It would also not be suited to warn about unusually high usage as the algorithm would adjust to the high usage and start only triggering when it gets extremely high. It would potentially be possible to have a very long memory length so that it would trigger for a while when a crash happens before it adjusts. This would however make the algorithm less sensitive. Due to its nature of triggering when something is different compared to what it has stored in memory, it it also prone to give false negatives and false positives.

This algorithm, namely the leap detection test works very well while looking for spikes in CPU utilization, but might not be suited for detecting crashes. When looking for and detecting a crash, it would presumably be better to have an algorithm or function which report the crash after the machines has gotten stuck. The algorithm used here would report the crash the moment it happens, but without knowing whether it is an actual crash or just a sudden short spike in CPU utilization.

4.5 Leap detection test in decision making context

Even tough the algorithm and prototype might not be the best suited for detecting crashes and anomalies, it does still give some feedback on how the virtual machine operates. It gives feedback on when the machine

gets under sudden heavy load and when a crash might have happened. This means that it would be possible to implement it together with a policy-based decision-making tool to some extent. If we recognize that the triggers from the prototype should not be used to force any sort of action on the machine, but should instead be used for other purposes.

One potential use case is to store snapshots of the machine and logs when a trigger appears. This can be useful for easily finding out what processes places the machine under heavy load and for recording its behavioral pattern. If this is done by a dashboard or by someplace that does not have access to the virtual machines logs and processes, it can still be used to try and figure out which machines puts the system under heavy load, and when to expect those heavy loads.

While the prototype might not be able to know when a crash has happened, we know from the third round of testing that it triggered multiple times in a row when it hits the plateau. It it therefore possible to have a policy-based decision-making tool notify users of potential crashes when consecutive triggers have been recorded. Although it might not be a crash, it still signals that at this point in time, something happened with the machine that might be worth looking into.

This means that there still are possible real life scenarios where combining the algorithm and prototype with a policy-based decision-making tool is beneficial. Although it is not a foolproof anomaly and crash detector, it is still usable to detect potential anomalies and crashes.

Chapter 5

Discussion

As mentioned in the Takeaway and Leap detection test in decision making context chapters, the prototype and algorithm might not be the most ideal fit for anomaly detection. The algorithm is after all based on a mathematical expression which is used to detect standard deviation, and as we know, deviation is not an uncommon occurrence in machines and virtual machines. The information gather from this study indicates that it might be more suitable to use another algorithm or approach when trying to detect crashes, resource hogging and other anomalies in virtual machines.

This does not however, mean that there are no scenarios where the algorithm and prototype could be useful. As mentioned in the previous chapter, the solution could still be used for detecting spikes in CPU usage which is still a useful metric. Due to its nature of triggering repeatedly upon hitting a plateau it could also be used to notify when a possible plateau or crash has appeared. It is also unique by being an algorithm which only observes CPU usage. This means that it can be used to observe virtual machines without being able view what goes on inside the machine. Since it only looks at CPU usage, it also has the potential to very lightweight being capable of running on most if not all systems.

Although the prototype could be improved and fine-tuned further, the results and findings so far indicates that further improvement on the prototype would most likely lead to a more fine tune CPU spike detector. To generalize it, the algorithm has use-cases as mentioned but is as mentioned not the best suited to detect and warn about abnormal behavior with high certainty.

Chapter 6

Conclusion

The testing and development of the algorithm and prototype over the course of this study revealed that while it was aimed at detecting crashes and anomalies in virtual machine, it fell short of this objective. The prototype which was designed to use the algorithm to monitor deviations in CPU usage, mostly succeeded in identifying short-term spikes in CPU usage without any reliable way of telling whether it was a crash or abnormal behavior. The tests showed that although the algorithm could be fine-tune, it would most likely still fall short of the original goal of being a reliable way of detecting crashes and abnormal behavior.

Although that is the case, there is still a potential use case for the algorithm as tool which observes CPU usage and reports on its pattern for other use-cases. It could also still be paired with a policy-based decision-making system to warn about CPU usage which could potentially be a crash.

The next step in this process and a topic of further research would be to investigate different algorithms to try and find one that would better handle the data and report on crashes with higher reliability.

Chapter 7

References

Bibliography

10 types of prototypes (with explanations and tips) [Indeed career guide]. (2023, April 11). Retrieved August 14, 2024, from https://www.indeed.com/career-advice/career-development/types-of-prototyping

Eucalyptus (software) [Page Version ID: 1214240797]. (2024, March 17). In *Wikipedia*. Retrieved May 8, 2024, from https://en.wikipedia.org/w/index.php?title=Eucalyptus_(software)&oldid=1214240797

The GNU general public license v3.0 - GNU project - free software foundation. (n.d.). Retrieved May 8, 2024, from https://www.gnu.org/licenses/gpl-3.0.html

Granat, Janusz, Batalla, Jordi Mongay, Mavromoustakis, Constandinos, & Mastorakis, George. (2019). Big data analytics for event detection in the IoT-multicriteria approach — IEEE journals & magazine — IEEE xplore. *IEEE Internet of Things Journal*, 4418–4430. Retrieved August 14, 2024, from https://ieeexplore.ieee.org/document/8920092

Grid computing [Page Version ID: 1205106912]. (2024, February 8). In *Wikipedia*. Retrieved February 25, 2024, from https://en.wikipedia.org/w/index.php?title=Grid_computing&oldid=1205106912

History of apache CloudStack — apache CloudStack. (n.d.). Retrieved March 30, 2024, from https://cloudstack.apache.org/history/

How to reduce your account CPU seconds usage? - knowledge base - ScalaHosting [Https://www.scalahosting.com/kb/] [Section: Software]. (2021, June 25). Retrieved May 5, 2024, from https://www.scalahosting.com/kb/how-to-reduce-your-account-cpu-seconds-usage/

Islam, M. S., & Miranskyy, A. (2020). Anomaly detection in cloud components [ISSN: 2159-6190]. *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 1–3. https://doi.org/10.1109/CLOUD49709.2020.00008

Liu, J., Zhang, H., & Xu, G. (2018). An anomaly detector deployment awareness detection framework based on multi-dimensional resources balancing in cloud platform [Conference Name: IEEE Access]. *IEEE Access*, *6*, 44927–44933. https://doi.org/10.1109/ACCESS.2018.2865114

OpenStack [Page Version ID: 1219141663]. (2024, April 16). In *Wikipedia*. Retrieved May 8, 2024, from https://en.wikipedia.org/w/index.php?title=OpenStack&oldid=1219141663

Resource utilization and performance. (2022, October 21). Retrieved April 29, 2024, from https://www.ibm.com/docs/en/informix-servers/14.10?topic=basics-resource-utilization-performance

Surbiryala, J., & Rong, C. (2019). Cloud computing: History and overview. *2019 IEEE Cloud Summit*, 1–7. https://doi.org/10.1109/CloudSummit47114.2019.00007

Utility computing [Page Version ID: 1193039590]. (2024, January 1). In *Wikipedia*. Retrieved February 25, 2024, from https://en.wikipedia.org/w/index.php?title=Utility_computing&oldid=1193039590

What is SaaS? - software as a service [Salesforce]. (n.d.). Retrieved March 17, 2024, from https://www.salesforce.com/in/saas/

Wibowo, S., & Deng, H. (2016). Evaluating the performance of cloud services: A fuzzy multicriteria group decision making approach. *2016 International Symposium on Computer, Consumer and Control (IS3C)*, 327–332. https://doi.org/10.1109/IS3C.2016.92

Chapter 8

Appendix

8.1 Algorithm and its code - first and second iteration

```
import argparse
import sys
import math

# The size of the short term memory

MEMORY_LENGTH = 0

CHI_LIMIT = 0

# The memory buffer:

memory_buff = []

# Average for computation:

avg = 0

# Average used to print

prt_avg = 0

# Sum:
```

```
sum = 0

# The index for the buffer-rotation:

index = -1

# The currently read data item

data = None

data_prev = None

# The result of the CHI squared computation:

chi = 0

tchi = 0

max_avg = 0

# counter

counter = 0

result = 0

highest_chi = 0

# Final string

output_string = ""

##########################
# Application specific variables. Not part of the algorithm

parser = argparse.ArgumentParser(description='Description of your script')
parser.add_argument('-v', action='store_true', help='Enable verbose mode')
parser.add_argument('-m', type=int, help='Specify memory length')
```

```
parser.add_argument('-c', type=int, help='Specify CHI_LIMIT')
parser.add_argument('-q', action='store_true', help='Enable quiet mode')
parser.add_argument('-a', action='store_true', help='Enable ALL_EVENTS')
parser.add_argument('-p', action='store_true', help='Enable optimization for
    plotter')
parser.add_argument('-max', action='store_true', help='Chi is max value in
    memory, use only one. The program defaults '
                                                    'to this setting')
parser.add_argument('-min', action='store_true', help='Chi is min value in
    memory, use only one')

args = parser.parse_args()

QUIET_CHECK = None

QUIET = 1 if args.q else None
VERBOSE = 1 if args.v else None
CHI_LIMIT = args.c
ALL_EVENTS = 1 if args.a else None
PLOTTER = 1 if args.p else None
MEMORY_LENGTH = args.m

if args.max:
    EXTRA_MODE = 1
elif args.min:
    EXTRA_MODE = 2
elif CHI_LIMIT != 0:
    EXTRA_MODE = 3
else:
    EXTRA_MODE = 1

if MEMORY_LENGTH is None:
    MEMORY_LENGTH = 10

strings_list = []


def verbose(message):
```

```
    if VERBOSE:
        strings_list.extend([message, "\n"])


def print_summary():
    global QUIET_CHECK
    if QUIET:
        if CHI_LIMIT and chi > CHI_LIMIT and not QUIET_CHECK:
            strings_list.append("1 \n")
            QUIET_CHECK = 1
            return
        return

    if ALL_EVENTS:
        strings_list.append(f"{counter} ")
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append("1\n")
        else:
            strings_list.append("0\n")
        return

    if VERBOSE:
        print_memory()
        strings_list.extend([f"chi({MEMORY_LENGTH}) = {chi} [{chi ** 2}]\n", f
            "Chi limit = {CHI_LIMIT}\n", f"new index: {index} (sum: {sum}) [avg
            : {prt_avg}]\n"])
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append(" *\n")

        global highest_chi
        if chi > highest_chi:
            highest_chi = chi

    if not VERBOSE and not PLOTTER:
        strings_list.append(f"chi: {chi}")
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append(" *\n")
        else:
```

```
            strings_list.append("\n")

    if PLOTTER and counter != 0:
        if CHI_LIMIT and chi < CHI_LIMIT:
            strings_list.append(f"{split_values[0]},{chi},,{diff}\n")
        elif CHI_LIMIT:
            strings_list.append(f"{split_values[0]},{chi},{chi},{diff}\n")


def find_max():
    max_num = 0
    for number in memory_buff:
        if number > max_num:
            max_num = number
    return max_num


def find_min():
    min_num = memory_buff[0]
    for number in memory_buff:
        if number < min_num:
            min_num = number
    return min_num


def find_avg():
    avg_num = 0
    for number in memory_buff:
        avg_num = avg_num + number

    return avg_num / len(memory_buff)


def print_memory():
    global output_string
    if VERBOSE:
        line1 = ""
        line2 = ""
```

```
        for i in range(len(memory_buff)):
            line1 += f"{i}--"
            line2 += f"{memory_buff[i]}--"
        strings_list.append(f"{line1}\n{line2}\n")


have_started = False

for line in sys.stdin:
    split_values = line.split(',')
    data = float(split_values[1].strip())

    if data != 0 and data_prev is not None:
        have_started = True
    if have_started:
        index = index + 1
        counter = counter + 1
        diff = data - data_prev

        verbose("\n---read data:-{}\n---Date:-{}\n".format(diff,
            split_values[0]))

        if (len(memory_buff)) < MEMORY_LENGTH:
            sum += diff
            avg += diff / (MEMORY_LENGTH + 1)
            prt_avg = avg
            verbose("inserting-{}-at-position-{}-(sum:-{})-[avg:-{}]\n".format
                (diff, len(memory_buff), sum, avg))
            memory_buff.append(diff)
        else:
            avg += diff / (MEMORY_LENGTH + 1)
            prt_avg = avg

            if EXTRA_MODE == 2:
                mem_min = find_min()
                min_avg = avg + mem_min / (len(memory_buff) + 1)
                tchi = math.sqrt((sum - MEMORY_LENGTH * mem_min) ** 2 / (
                    MEMORY_LENGTH * (MEMORY_LENGTH + 1) * min_avg))
```

```
                CHI_LIMIT = tchi * 2

            elif EXTRA_MODE == 1:
                mem_max = find_max()
                max_avg = avg + mem_max / (len(memory_buff) + 1)
                tchi = math.sqrt((sum - MEMORY_LENGTH * mem_max) ** 2 / (
                    MEMORY_LENGTH * (MEMORY_LENGTH + 1) * max_avg))
                CHI_LIMIT = tchi * 2
            # lage en funsjon som endrer CHI_LIMIT basert p  forskjellige
                args in commandline

            chi = math.sqrt((sum - MEMORY_LENGTH * diff) ** 2 / (MEMORY_LENGTH
                * (MEMORY_LENGTH + 1) * avg))
            index = index % MEMORY_LENGTH
            sum = sum - memory_buff[index] + diff
            avg = avg - (memory_buff[index] / (MEMORY_LENGTH + 1))
            memory_buff[index] = diff
        if CHI_LIMIT and chi > CHI_LIMIT:
            result += 1

    data_prev = data
    print_summary()

finished = "".join(strings_list)
if VERBOSE:
    print(f"{result},{counter}\nHighest chi is = {highest_chi}\n\n" + finished
        )
else:
    print(finished)
```

8.2 Algorithm and its code - third iteration

```
import argparse
import sys
import math

# The size of the short term memory
```

```
MEMORY_LENGTH = 0

CHI_LIMIT = 0

# The memory buffer:

memory_buff = []

# Average for computation:

avg = 0

# Average used to print

prt_avg = 0

# Sum:

sum = 0

# The index for the buffer-rotation:

index = -1

# The currently read data item

data = None

data_prev = None

# The result of the CHI squared computation:

chi = 0

tchi = 0

max_avg = 0
```

```
# counter

counter = 0

result = 0

highest_chi = 0

# Final string

output_string = ""

########################
# Application specific variables. Not part of the algorithm

parser = argparse.ArgumentParser(description='Description of your script')
parser.add_argument('-v', action='store_true', help='Enable verbose mode')
parser.add_argument('-m', type=int, help='Specify memory length')
parser.add_argument('-c', type=int, help='Specify CHI_LIMIT')
parser.add_argument('-q', action='store_true', help='Enable quiet mode')
parser.add_argument('-a', action='store_true', help='Enable ALL_EVENTS')
parser.add_argument('-p', action='store_true', help='Enable optimization for
    plotter')
parser.add_argument('-max', action='store_true', help='Chi is max value in
    memory, use only one. The program defaults '
                                                    'to this setting')
parser.add_argument('-min', action='store_true', help='Chi is min value in
    memory, use only one')

args = parser.parse_args()

QUIET_CHECK = None

QUIET = 1 if args.q else None
VERBOSE = 1 if args.v else None
CHI_LIMIT = args.c
ALL_EVENTS = 1 if args.a else None
```

```
PLOTTER = 1 if args.p else None
MEMORY_LENGTH = args.m

if args.max:
    EXTRA_MODE = 1
elif args.min:
    EXTRA_MODE = 2

elif CHI_LIMIT != 0:
    EXTRA_MODE = 3
else:
    EXTRA_MODE = 1

if MEMORY_LENGTH is None:
    MEMORY_LENGTH = 10

strings_list = []


def verbose(message):
    if VERBOSE:
        strings_list.extend([message, "\n"])


def print_summary():
    global QUIET_CHECK
    if QUIET:
        if CHI_LIMIT and chi > CHI_LIMIT and not QUIET_CHECK:
            strings_list.append("1-\n")
            QUIET_CHECK = 1
            return
        return

    if ALL_EVENTS:
        strings_list.append(f"{counter}-")
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append("1\n")
        else:
```

```
            strings_list.append("0\n")
        return

    if VERBOSE:
        print_memory()
        strings_list.extend([f"chi({MEMORY_LENGTH}) = {chi} [{chi ** 2}]\n", f
            "Chi limit = {CHI_LIMIT}\n",
                             f"new index: {index} (sum: {sum}) [avg: {prt_avg
                                 }]\n"])
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append(" *\n")

    if not VERBOSE and not PLOTTER:
        strings_list.append(f"chi: {chi}")
        if CHI_LIMIT and chi > CHI_LIMIT:
            strings_list.append(" *\n")
        else:
            strings_list.append("\n")

    if PLOTTER and counter != 0:
        if CHI_LIMIT and chi < CHI_LIMIT:
            strings_list.append(f"{split_values[0]},{chi},,{diff}\n")
        elif CHI_LIMIT:
            strings_list.append(f"{split_values[0]},{chi},{chi},{diff}\n")


def find_max():
    max_num = 0
    for number in memory_buff:
        if number > max_num:
            max_num = number
    return max_num


def find_min():
    min_num = memory_buff[0]
    for number in memory_buff:
        if number < min_num:
```

```
            min_num = number
    return min_num


def find_avg():
    avg_num = 0
    for number in memory_buff:
        avg_num = avg_num + number

    return avg_num / len(memory_buff)


def print_memory():
    global output_string
    if VERBOSE:
        line1 = ""
        line2 = ""
        for i in range(len(memory_buff)):
            line1 += f"{i} "
            line2 += f"{memory_buff[i]} "
        strings_list.append(f"{line1}\n{line2}\n")


have_started = False

for line in sys.stdin:
    split_values = line.split(',')
    data = float(split_values[1].strip())
    if data != 0 and data_prev is not None:
        have_started = True
    if have_started:
        index = index + 1
        counter = counter + 1
        diff = data - data_prev

        verbose("\n--- read data: {}\n--- Date: {}\n".format(diff,
            split_values[0]))
```

```
    if (len(memory_buff)) < MEMORY_LENGTH:
        sum += diff
        avg += diff / (MEMORY_LENGTH + 1)
        prt_avg = avg
        verbose("inserting {} at position {} (sum: {}) [avg: {}]\n".format
            (diff, len(memory_buff), sum, avg))
        memory_buff.append(diff)
    else:
        avg += diff / (MEMORY_LENGTH + 1)
        prt_avg = avg

        if counter < 7500:
            mem_max = find_max()
            max_avg = avg + mem_max / (len(memory_buff) + 1)
            tchi = math.sqrt((sum - MEMORY_LENGTH * mem_max) ** 2 / (
                MEMORY_LENGTH * (MEMORY_LENGTH + 1) * max_avg))
            CHI_LIMIT = tchi * 2

        elif counter == 7500:
            rounded_highest_chi = math.floor(highest_chi)
            CHI_LIMIT = rounded_highest_chi

        chi = math.sqrt((sum - MEMORY_LENGTH * diff) ** 2 / (MEMORY_LENGTH
            * (MEMORY_LENGTH + 1) * avg))

        if chi > highest_chi:
            highest_chi = chi

        index = index % MEMORY_LENGTH
        sum = sum - memory_buff[index] + diff
        avg = avg - (memory_buff[index] / (MEMORY_LENGTH + 1))
        memory_buff[index] = diff
    if CHI_LIMIT and chi > CHI_LIMIT:
        result += 1

data_prev = data
print_summary()
```

```
finished = "".join(strings_list)
if VERBOSE:
    print(f"{result},{counter}\nHighest chi is = {highest_chi}\n\n" + finished
        )
else:
    print(finished)
```

8.3 1. Script used to run main.py and organize file structure

```
#!/bin/bash

# Initialize variables with default values
current_date=$(date +%Y-%m-%d-%H-%M-%S)
max_flag=""
min_flag=""
avg_flag=""
verbose_flag=""

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    key="$1"

    case $key in
        -p)
        # Set the flag for -p
        plotter_flag="-p"
        ;;
        -v)
        # Set the flag for -v
        verbose_flag="-v"
        ;;

        -max)
        # Set the flag for -max
        max_flag="-max"
        ;;
        -min)
        # Set the flag for -min
```

```
        min_flag="-min"
        ;;
        -avg)
        # Set the flag for -avg
        avg_flag="-avg"
        ;;
        *)
        # Unknown option
        ;;
    esac
    shift
done
current_date=$(date +%Y-%m-%d-%H-%M-%S)
mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag"
cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag" ||
    exit
for ((i=0; i<=223; i++))
do
  formatted_number=$(printf "%04d" $i)

  output_file="VM-$formatted_number.txt"
  echo "$formatted_number"

  awk -F ',' '$3 ~ /VM-'"$formatted_number"'/' ../result-timestamp-changed.csv
      | cut -d ',' -f 2,1 | python3 ../main.py $max_flag $min_flag $avg_flag
      $verbose_flag $plotter_flag > "$output_file"

done
```

8.4 2. Script used to run the memory test and organize file structure

```
#!/bin/bash

# Initialize variables with default values
current_date=$(date +%Y-%m-%d-%H-%M-%S)
max_flag=""
min_flag=""
```

```
avg_flag=""
verbose_flag=""

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    key="$1"

    case $key in
        -p)
        # Set the flag for -p
        plotter_flag="-p"
        ;;
        -v)
        # Set the flag for -v
        verbose_flag="-v"
        ;;
        -max)
        # Set the flag for -max
        max_flag="-max"
        ;;
        -min)
        # Set the flag for -min
        min_flag="-min"
        ;;
        -avg)
        # Set the flag for -avg
        avg_flag="-avg"
        ;;
        *)
        # Unknown option
        ;;
    esac
    shift
done
current_date=$(date +%Y-%m-%d-%H-%M-%S)
mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag+
    _memory_test"
cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag+
```

```
    _memory_test" || exit
for ((i=5; i<=30; i++))
do
  formatted_number=$(printf "%s" "$i")

  output_file="VM-0008_mem_$formatted_number.txt"
  echo "$formatted_number"

  awk -F ',' '$3 ~ /VM-0008/' ../result-timestamp-changed.csv | cut -d ',' -f
      2,1 | python3 ../main.py $max_flag $min_flag $avg_flag $verbose_flag
      $plotter_flag -m $formatted_number > "$output_file"

done
```

8.5 2. Script used to run the third iteration called adaptive_main.py and organize file structure

```
#!/bin/bash

# Initialize variables with default values
current_date=$(date +%Y-%m-%d-%H-%M-%S)
max_flag=""
min_flag=""
avg_flag=""
verbose_flag=""

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    key="$1"

    case $key in
        -p)
        # Set the flag for -p
        plotter_flag="-p"
        ;;
        -v)
        # Set the flag for -v
        verbose_flag="-v"
```

```
        ;;
        -max)
        # Set the flag for -max
        max_flag="-max"
        ;;
        -min)
        # Set the flag for -min
        min_flag="-min"
        ;;
        -avg)
        # Set the flag for -avg
        avg_flag="-avg"
        ;;
        *)
        # Unknown option
        ;;
    esac
    shift
done
current_date=$(date +%Y-%m-%d-%H-%M-%S)
mkdir "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag+
    -adaptive"
cd "$current_date$max_flag$min_flag$avg_flag$verbose_flag$plotter_flag+
    -adaptive" || exit

formatted_number=$(printf "%s" "30")

output_file="VM-0008_adaptive.txt"
echo "$formatted_number"

awk -F ',' '$3 ~ /VM-0008/' ../vm-0008-orig-csv.csv | cut -d ',' -f 2,1 |
    python3 ../adaptive_main.py $max_flag $min_flag $avg_flag $verbose_flag
    $plotter_flag -m 30 > "$output_file"
```

8.6 Distribution plot script

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import os
import sys
import pandas as pd

# Function to parse the first line of a file and extract the relevant
    information
def parse_first_line(file_path):
    with open(file_path, 'r') as file:
        first_line = file.readline().strip()
        if first_line == '':
            return None
        value1, value2 = map(int, first_line.split(','))
    return value1, value2

# Function to extract data from a directory
def extract_data(directory):
    data = []
    for filename in os.listdir(directory):
        if filename.startswith("VM-") and filename.endswith(".txt"):
            file_path = os.path.join(directory, filename)
            values = parse_first_line(file_path)
            if values:
                data.append(values)
    return pd.DataFrame(data, columns=['Value1', 'Value2'])

# Function to plot the distribution from two directories
def plot_distributions(directory1, directory2, output_path):
    # Extract data from both directories
    df1 = extract_data(directory1)
    df2 = extract_data(directory2)

    # Calculate the ratios
    ratios1 = df1['Value1'] / df1['Value2']
    ratios2 = df2['Value1'] / df2['Value2']

    # Calculate the common range for both histograms
    min_value = min(ratios1.min(), ratios2.min())
    max_value = max(ratios1.max(), ratios2.max())
```

```
    # Plotting the distribution
    plt.figure(figsize=(10, 6))
    bins = 20  # Set a common bin size for both histograms

    # Plot the first histogram
    plt.hist(ratios1, bins=bins, alpha=0.5, label='Min function', edgecolor='
        blue', histtype='stepfilled', color='lightblue', range=(min_value,
        max_value), density=True)

    # Plot the second histogram
    plt.hist(ratios2, bins=bins, alpha=0.5, label='Max function', edgecolor='
        red', histtype='stepfilled', color='pink', range=(min_value, max_value)
        , density=True)



    plt.xlabel('Triggers / Lines')
    plt.ylabel('Frequency')
    plt.title('Distribution of Triggers / Lines')
    plt.legend(loc='upper right')
    plt.grid(True)
    plt.savefig(output_path)
    plt.show()

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: python script.py /path/to/first/source/ /path/to/second/
            source/ output.png")
        sys.exit(1)

    directory1 = sys.argv[1]
    directory2 = sys.argv[2]
    output_path = sys.argv[3]
    plot_distributions(directory1, directory2, output_path)
```

8.7 Plot script

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
import argparse

def plot_data(file_path, title):
    # Read the data
    data = pd.read_csv(file_path, header=None, names=['X', 'Y1', 'Y2', 'Y3'])

    # Create a plot and twin axes
    fig, ax1 = plt.subplots(figsize=(10, 6))
    ax2 = ax1.twinx()

    # Plotting the first column against the second column
    ax1.plot(data['X'], data['Y1'], label='Line-plot', linestyle='-')

    # Plotting the third column with X markers
    ax1.plot(data['X'], data['Y2'], label='X-markers', linestyle='None',
        marker='x')

    # Plotting the fourth column on the secondary axes
    ax2.plot(data['X'], data['Y3'], color='lightgrey')

    # Filling the area beneath the Y3 line with a weak background color
    ax2.fill_between(data['X'], data['Y3'], color='lightgrey', alpha=0.3)

    # Adding title and labels
    ax1.set_title(title)
    ax1.set_xlabel('X-axis')
    ax1.set_ylabel('Chi-value', color='tab:blue')
    ax2.set_ylabel('CPU-value', color='lightgrey')
    ax1.legend()

    # Adjust x-axis ticks and labels
    x_ticks = data['X'][::int(len(data['X']) / 10)]  # Adjust the interval
        here
    ax1.set_xticks(x_ticks)
    ax1.set_xticklabels(data['X'][::int(len(data['X']) / 10)], rotation=45, ha
        ='right')  # Adjust rotation and alignment
```

```
    # Adjust subplot parameters to leave space for x-axis labels
    plt.subplots_adjust(bottom=0.2)

    # Save the plot as a PNG file
    plt.savefig(f"{title}.png")

    # Show the plot
    plt.show()

if __name__ == '__main__':
    # Setup command line argument parsing
    parser = argparse.ArgumentParser(description="Plot data from a text file.")
    parser.add_argument("file_path", type=str, help="Path to the text file
        containing the data.")
    parser.add_argument("title", type=str, help="Title of the plot.")
    args = parser.parse_args()

    # Call the plot function
    plot_data(args.file_path, args.title)
```

8.8 Memory plot for triggers

```
import os
import matplotlib.pyplot as plt

# Assuming your files are named sequentially and are in the same folder
folder_path = '2024-06-17-19-37-48-max-v+_memory_test-max'  # Replace with
    your folder path
memory_lengths = range(5, 31)
y_values = []

for memory_length in memory_lengths:
    file_name = f'VM-0008_mem_{memory_length}.txt'  # Adjust this based on
        your actual file naming convention
    file_path = os.path.join(folder_path, file_name)

    with open(file_path, 'r') as file:
```

```
            first_line = file.readline().strip()
            # Split the line by comma and extract the first number
            y_value = float(first_line.split(',')[0])
            y_values.append(y_value)

# Now plot the data
plt.plot(memory_lengths, y_values, marker='o')
plt.title('Number of triggers based on memory length')
plt.xlabel('Memory Length')
plt.ylabel('Number of triggers')
plt.grid(True)
plt.show()
```

8.9 Memory plot for chi-value

```
import os
import matplotlib.pyplot as plt

# Folder path where your files are located
folder_path = '2024-06-17-19-37-48-max-v+_memory_test-max'  # Update with your
    actual folder path

# Define the memory lengths range
memory_lengths = range(5, 31)
chi_values = []

for memory_length in memory_lengths:
    file_name = f'VM-0008_mem_{memory_length}.txt'  # File naming convention
    file_path = os.path.join(folder_path, file_name)

    with open(file_path, 'r') as file:
        lines = file.readlines()
        # Look for the line that contains "Highest chi is = "
        for line in lines:
            if "Highest chi is =" in line:
                # Extract the number after "Highest chi is ="
                chi_value = float(line.split("Highest chi is =")[1].strip())
                chi_values.append(chi_value)
```

```
                    break

# Now plot the data
plt.plot(memory_lengths, chi_values, marker='o')
plt.title('Highest chi value based on memory length')
plt.xlabel('Memory Length')
plt.ylabel('Highest Chi Value')
plt.grid(True)
plt.show()
```