Clustering Analysis of Lock Contention Fault Types Using Run-time Performance Metrics for Java Intrinsic Locks

by

Nahid Hasan Khan

A thesis submitted to the

School of Graduate and Postdoctoral Studies in partial

fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

January 2022

Abstract

Locks are essential in java-based multi-threaded applications as this mechanism provides a proper solution to synchronizing shared resources. However, improper management of locks and threads can lead to contention; as a result, it causes performance degradation and prevent java application from further scaling. These types of faults are challenging to debug because they are caused by complex interactions among the threads and can only be detected at run time. Nowadays, performance engineers use legacy tools and their experience to determine causes of lock contention. In this research, a clustering-based approach is presented to help identify the type of lock contention fault to facilitate the procedure that performance engineers follow, intending to support developers with less experience eventually. The classifier is based on the premise that if lock contention exists it is reflected as either a) threads spend too much time inside the critical section and/or b) threads' high frequency access to the locked resources. Our results show that a classifier can be effectively trained to detect lock contention caused by high hold time and contention due to high frequency with which threads send access requests to the locked resources.

Keywords— Java, Locks, Monitors, Contention, Concurrency, Run-time Faults, Localisation, JLM, Linux Perf, Clustering, Classification, Software Engineering

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Acknowledgments

I would like to acknowledge everyone who played a role in my academic accomplishments. First of all, my mother and my father. Without you, I could never have reached this current level of success.

Secondly, my excellent Supervisors, Drs. Akramul Azim and Ramiro Liscano for providing me the opportunity to work in RTEMSOFT Research Lab at Ontario Tech University, where I have conducted this research work. I would like to thank them for providing proper guidelines and supporting me in every phase of this academic journey. Thanks to Vijay Sundaresan and Yee-Kang Chang from IBM Team for providing me with the appropriate guidelines and the fruitful discussion we had in the meetings. I would like to thank my project-mate, Joseph Robertson, who is continuously working with me on this same project and has made an outstanding contribution to this research work.

Last but not least, I would like to thank my committee members, each of whom has provided patient advice and guidance throughout the research process. Thank you all for your support.

Statement of Contributions

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 INTRODUCTION

The Java programming language has multi-threading capabilities for concurrent programming. It provides an Application Programming Interface (API) for managing multi-threaded concurrency processing known as an intrinsic lock or monitor lock that observes the behaviour of threads and enforces exclusive access to any object's state [1]. Every object in java has an inherent lock that monitors the threads' movements trying to access shared resources. However, the Java Virtual Machine implements this built-in locking mechanism internally that provides the opportunity of thread synchronization for concurrent programming [2].

Synchronization in java language provides an efficient solution to access shared resources by multiple threads and avoid data inconsistency. Although synchronization is essential in multi-threaded applications, it introduces some levels of thread contention when applied. Multiple threads block other threads when two or more threads try to access the same shared resources in a thread contention scenario. These multiple threads then undergo a slow operation and sometimes even suspend execution entirely. As a result, it can turn the application to perform poorly in a multi-threaded application. This performance degradation is typically known as a contention fault or performance bottleneck due to contention.

It is also known that contention faults are caused by two primary issues [3] described by Brian Goetz in his book that are:

- Type 1 - Threads spend too much time inside the critical sections.
- Type 2 - High frequency with which threads send access requests to the locked resources.

In a critical section, introducing few additional and unnecessary computations makes one thread hold the lock longer than expected. This indicates the first type of fault. According to Goetz, if an operation holds a lock for more than equal to two milliseconds then no matter how many idle processors are there, the throughput of the application never exceed five hundred operations per second [3]. On the other hand, fault type 2 occurs when many shared resources are tightened up in a single lock, and they are accessed by an increased number of threads. Therefore these threads increase the request frequency to the locked resources. For example, two shared resources resource-1 and resource-2, are locked in a single lock. Thread-1 and Thread-2 need to access resource-1 only, and Thread-3 and Thread-4 need to modify resource-2. The request frequency is gone high when these four threads try to access the single lock simultaneously. The two shared resources can be locked in two different lock objects, which usually decreases the request frequency.

Leaving these types of patterns in a concurrent code-base creates performance bottlenecks, which is difficult to find using any static analysis approach. Therefore, run-time metrics help developers identify the actual cause for contention hidden at the code level. With this in mind we are interested in developing a contention classifier that assists in identifying contention fault types throwing some proper recommendations.

It is hard to write concurrent programs and developers usually come back to refactor the portion of the code where the concurrency feature resides to make their concurrent code more efficient. A recent study reports that more than 25% of all critical sections are changed at some point by the developers, both to fix correctness bugs and to enhance performance [4], [5]. The motivation for our work is to automate contention fault detection and identification by leveraging the fact that there are 2 potential causes for contention faults as described by Goetz [3].

Even though contention bottlenecks have been investigated in the software community for a while they are still difficult to detect [6], [7] and analyze and usually it is a job performed by an experienced performance engineer. Typically application developers do not have the skill set that a performance engineer has. To detect contention bottlenecks, performance engineers usually use some legacy tools such as IBM Performance Inspector [8], YourKit Java Profiler [9], JProfiler [10] etc.

In this research, we focus on contentions caused by the improper use of java intrinsic locks. Improper use of intrinsic lock implies leaving two types of harmful patterns in the concurrent section of the codebase. We use a run-time analysis approach over static code analysis because these faults surface at run-time. The analysis is based on performance metrics such as GETS, TIER2, TIER3, AVER_HTM are collected from Java Lock Monitor (JLM) [8] and metrics such as "_raw_spin_lock", "ctx_sched_in" that are collected from *perf* [11] analyzer tool. The preliminary research of ours has found that these fault types tend to leave some patterns in the run-time logs depending on their behaviors. Therefore, we believe that it is possible to classify contention fault types into these two causes using a clustering approach that will identify the essential features from the JLM and *perf* run-time metrics.

1.2 Research Questions

This research experiences some primary questions regarding contention classification using the clustering ML approach. At the end of this thesis, our study tries to answer these questions. The questions are:

1. How is this method good enough over traditional approaches?
2. Why ML is needed for this type of work?

1.3 Motivation

The traditional approaches [12] [13] [14] [15] [9] [10] [16] fail to distinguish the contention fault types and hence it is challenging to produce recommendations or suggestions about the cause of the contention. These approaches present different thread activities, blocked thread lists, and lock-monitor statistics that do not provide insight into these two faults. Therefore, developers struggle to identify the actual reason for faults and can not distinguish which fault needs to be addressed. In addition, it is difficult to check the fault's reason manually at run-time. However, performance metrics (e.g., GETS, TIER2, TIER3, AVER_HTM, _raw_spin_lock etc) can benefit developers from identifying the fault types during the run-time.

Therefore, we propose to automate the fault identification using machine learning from the contention metrics. In our approach, we create a dataset considering different features that are responsible for creating these faults. We then propose a clustering approach to classify the fault types so that developers can easily recognize and resolve the faults.

Moreover, we find a connection between the extracted clusters and their usefulness. The extracted clusters can directly benefit the engineers or developers by narrowing down the problem scope. Based on these two types of clusters, developers can apply some solutions that are discussed below:

1.3.1 Recommended solutions for fault type 1:

Following solutions are recommended based on the situation after a cluster indicates the fault type 1. These are:

1. Omitting expensive operations or calculations in the critical sections that consume extra execution times can reduce holding the lock for long. Expensive operations that is not related to shared state of the object should not be guarded by the lock. We need to protect the shared state of the object, not the code. Example of this particular faulty pattern and the solution is

Listing 1 Time-consuming fault pattern in concurrent code in java application

```java
public final HashSet<Object> tasks;
public void taskOne(Object val){
    synchronized(lock1){
        Object val1 = someProcessing(val); // unnecessary codes
        tasks.add(val1);
    }
}
```

Listing 2 Solution of the time-consuming fault pattern in concurrent java application

```java
public final HashSet<Object> tasks;
public void taskOne(Object val){
    // leave unnecessary code out of the critical section
    Object val1 = someProcessing(val);
    synchronized(lock1){
        tasks.add(val1);
    }
}
```

shown in the Listings of Listing 1 and Listing 2 respectively.

2. Avoiding synchronized method can be another solution to reduce hold time. Using synchronized method guards all the content inside it. Therefore, holding time increases and best practice is to apply synchronized block and guard the shared resource only. This type of pattern can be shown for example, the code in Listing 3. In order to avoid this faulty situation, it is recommended to follow the solution pattern, which is shown in Listing 2.

3. Applying read-write lock reduces the execution time in a critical section. The approach of read-write lock implements open to read (unless no writing is in progress or write request) but close to write. Multiple threads have the permission to read simultaneously as long as any thread does not attempt to write at that moment, or there is no incoming write request. On the other hand the writing mechanism still follow the mutual exclusion fashion. Java provides API for the read-write lock and can be achieved through

Listing 3 Synchronized method fault pattern in concurrent code in java application

```
public synchronized void taskOne(Object val){
    Object val1 = someProcessing(val);
    tasks.add(val);
}
```

Listing 4 ReadWriteLock solution pattern in concurrent code in java application

```
readLock.lock();
try {
    tasks.get(val);
}
finally {
   readLock.unlock();
}

writeLock.lock();
try {
    val1 = someProcessing(val);
    tasks.add(val1);
}
finally {
    writeLock.unlock();
}
```

java.util.concurrent.ReadWriteLock package. The read-write lock pattern utilizing the package is shown in Listing 4

4. Java provides a package **java.util.concurrent** (JUC) that ensures efficient and thread-safe concurrency. Leveraging **java.util.concurrent** package and its containers extra amount of execution time under a critical section can be reduced. The recommended practice is to utilize concurrent containers (e.g., synchronizedHashMap, synchronizedHashSet etc) rather than classical data-structure (e.g., HashMap, HashSet etc) in case concurrency is required in a java application.

Listing 5 Splitting lock approach to reduce access frequency by threads in concurrent code in java application

```java
public final HashSet<Object> tasks1;
public final HashSet<Object> tasks2;

// problematic pattern
public void taskOne(Object val1, Object val2){
    // leave unnecessary code out of the critical section
    Object val1 = someProcessing(val);
    Object val2 = claculation(val2);
    synchronized(lock1){
        tasks1.add(val1);
        tasks2.add(val2);
    }
}

// solution pattern
public void taskOne(Object val){
    // leave unnecessary code out of the critical section
    Object val1 = someProcessing(val);
    synchronized(lock1){
        tasks.add(val1);
    }
}

public void taskTwo(Object val2){
    // leave unnecessary code out of the critical section
    Object val2 = calculation(val2);
    synchronized(lock2){
        tasks.add(val2);
    }
}
```

1.3.2 Recommended solutions for fault type 2:

Once a cluster appear with type 2 the following recommended solutions can be applied in order to reduce the contention.

1. Suggested solution #2 “Avoid using synchronized method” from above fault type 1 is also applicable for fault type 2. Due to the synchronized method, a lock guards the whole class object and is inaccessible to others. Hence, the lock acquisition metrics or spinning counts around a lock are increased when the thread number increases and threads send the access request with high frequency. Solution for avoiding using synchronized method is shown in Listing 2.

2. The lock splitting approach is also efficient for a competitive lock. Instead of guarding multiple independent state variables, splitting the lock into multiple locks is recommended. With such change, it enhances the performance by reducing the lock competition with which locks send access requests to the locks. An example solution for splitting a lock is shown in Listing 5.

3. Similar to lock splitting, stripping a lock helps reduce the competition significantly for acquiring a lock. In this fashion, an independent state variable is separated into many blocks that will be guarded by some set of locks that ensures low lock competition with efficiency and enhanced scalability. An example of lock stripping can be found in Brian Goetz’s book in Chapter 11 Section 4 [3].

4. Implementing the technique read-write lock can also be a solution in order to reduce lock competition which is shown already in Listing 4.

5. Leveraging **java.util.concurrent** package and its containers could be another tip to solve the problem occurring due to fault type 2. Concurrent containers implement the stripping approach internally that helps reduce the contention greatly.

1.4 Contributions

The main contributions of our research are:

1. Classifying contention fault types of java-based concurrent application through clustering techniques utilizing the run-time metrics that come from performance analyzer tools.
2. Generation of a dataset containing contention statistics and formalization of the experiments so that by leveraging this formalization one can enrich dataset with new sets of contention faults.

1.5 Organization of the Thesis

Our research ~~paper~~ is organized by the following chapters. In Chapter 2 we introduce the reader to some related works discussing their approaches. There are quite a few related works listed that have been dealing with java performance degradation due to contention bottlenecks. We end the chapter by introducing the readers to the current traditional approaches that are being used to analyze lock contention faults or bugs. It describes how performance engineers operate IBM Performance Inspector and what the steps are, how performance engineers deal with some other popular tools such as YourKit, JProfiler, VisualVM, JDK utils etc. We also list the current approaches' limitations at the end of this chapter.

We continue in Chapter 3 where we present our methodology for our approach. First, we present a high-level workflow of our approach, then we try to explain the three main method steps that are essential to classify the lock contention faults. The very first step describes performance metrics acquisition secondly, metrics aggregation and filtering, and lastly, how we perform data preprocessing and classification.

In Chapter 4 we try to present the dataset generation process and the environment we set up for the experiment, and it has the details of both the hardware and software configuration. In terms of software configuration, it describes the java version and JVM we use and the tools we installed to capture performance metrics and continue our work. The chapter includes a detailed explanation of the log generation process and how we perform an automated generation process. Later this chapter, we detail the test formalization where we describe how the example code was configured to exercise and what parameters we changed during the experiment. In the end, the dataset information is discussed.

We present the clustering results in Chapter 5. In this chapter, we discuss the high-level observation of JLM and perf data with different example code configurations. We analyze the heatmap of the correlation matrix output and analyze essential features for our dataset. Before moving forward to the clustering process, in this chapter, we also try to show some verification and validation using popular R and Python packages that the data we generated is cluster-able. After that, we continue our work applying KMeans, PCA, and DBSCAN algorithms, and before that, we verify the actual cluster number using silhouette coefficient and elbow method. Finally, we end our chapter by evaluating the performance of our model.

Our next Chapter 6 discusses the mechanism we introduce to label the fault types leveraging the test parameters such as the number of threads and sleep time we record during the experiment. Plotting the threads and sleep time help us to label the fault types. The end of this chapter illustrates the advanced analysis of the metrics that are useful for further classification. And it has a short discussion about why the analysis and dominant metrics are crucial to the performance engineer as well as developers.

Our final Chapter 7 has the full overview of our work. We include the limitations of our current work and lastly it is ended with discussing the future work that we have in our bucket list.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 INTRODUCTION

Analyzing lock contention performance issues and locating and resolving them is an on-going research topic and has been investigated by developers and researchers for more than a decade. Because of the independent threads and their movements, it is hard to detect the locking issues, and it is more problematic when there are more independent state variables to be locked. Moreover, these lock-related issues can only be detected at run-time. Many approaches and tools have been published in order to deal with resolving contention and java performance degradation due to contention. Most of them discuss identifying contention or critical section pressure, and some of them have dealt with detecting and locating the contention region. Although these methods are extensive and efficient, they still fail to discuss the two fault types, which is holding the critical section for a long period of time and high frequent requests to the locked resources by threads. Moreover, these approaches lack analyzing the contention statistics and performing any classification process as we present in this work. However, in this chapter, we divide the content into two main sections a) discussing some approaches to deal with contention issues, b) discussing some popular methods and tools that have the solution to resolve contention and performance degradation.

2.2 Related Work

Lock contention performance bottlenecks have been investigated in the the past few years with researchers primarily focusing on detecting and locating the root cause of the lock contention Very few papers though have attempted to categorize the lock contention as we are investigating.

Nathan R. Tallent et al. [17] detail three approaches to gaining insight into performance losses due to lock contention. Their first two approaches used call stack profiling and proved that this profiling does not yield insight into lock contention. The final approach used an associated lock contention attribute called thread spinning that helps yielding insight into lock contention. Although the paper's analysis is based on "C" concurrent programs, their approaches are similar to ours. We are also considering run-time logs for the analysis and determining run-time metrics directly related to the contention fault and impacting the bottlenecks.

Another similar paper by Peter Hofer et al., [18], proposed a novel approach to detect lock contention in a Java application by tracing the locking events extracted from the JVM. Tracing call chains of both the blocked and blocking threads confirms the contention causes and severity of the running application. The main difference between their work and ours is the metrics we extracted from the run-time traces that bear the potential weight for contention severity measurement.

Florian David et al. proposed a profiler named "free-lunch" that measures critical section pressure (CSP) and the progress of the threads that impede the performance [12]. In this work, they modified the Java Virtual Machine Tool Interface (JVMTI) and captured the thread progression. This paper also stated that they failed to determine the correlation among the metrics extracted from IBM Java Lock Analyzer (JLA) while we have been able to observe some relations between the performance metrics and the lock contention. This paper also lacks a description of the metrics related to different contention fault types.

A different style of approach was proposed by E. Farchi et al. [13] where they create or find patterns that describe issues in the code and attempt to match those to real code examples. The paper used a tool called "ConTest" to test their assumption. They found that their system was able to enhance the "ConTest" tool's ability to locate concurrent bugs.

Sangmin Park et al. [19] proposes a tool named FALCON that dynamically analyses concurrent programs and attempts to locate problematic data-access patterns based on memory-access sequences among threads. It does this by observing memory access during the code execution and assigns them a pass or fail based on the pattern, the pass/fail ratio is then used to calculate a suspicion rating of the code. The tool is different from others because it captures both order violations and atomic violations.

R. Gopalakrishnan et al.[20] proposed a system that identifies problems in code structure and is able to provide a solution without having to first execute the code. It uses machine learning and text mining algorithms to mine the source code multiple open-source projects and identify the "source code topics" which are correlated with architectural tactics, these are then used to predict what the program should have based on the requirements that the machine learning model predicts. This is not the type of classification that we are using, however it is a interesting approach to the problem.

Chen Zhang et al. [14] implemented a static synchronization performance bug detection tool that detects critical section identifier, loop identifier, inner loop identifier, expensive loop identifier, and pruning component. They collected 26 performance bugs from three real-world distributed systems HDFS, Hadoop MapReduce, and HBase, to detect performance bugs, and their detection tool performed well on these. The main difference between this method and our is static and dynamic analysis. Moreover, they did not analyze the log trace and classify the fault types.

The IBM Health Center [15], a powerful tool, is built for internal use and quite good enough to deal with detecting the lock contention in a java-based application.JLM is listed under this tool but it requires manual observation and intervention to detect and locate the contention related bottlenecks.

2.3 Traditional Approaches

In our related work section, we attached several traditional approaches that analyze lock contention bottlenecks, but none had ever gone with the clustering approach such as ours. The benefits of clustering techniques are many. It reduces human intervention, reveals insight into the contention-related performance metrics, reveals new classes of fault types. However, there are several lock contention monitoring tools & techniques published there. Among them, some popular tools are widely used, such as "IBM Performance Inspector" [8], "YourKit Java Profiler" [9] etc. This section intends to go through these tools and their approaches for detecting lock contention bottlenecks in case ~~of~~ contention occurs.

2.3.1 IBM Performance Inspector

IBM Performance Inspector is a performance benchmark-suite built for internal use and publicly inaccessible. Tools such as JPROF, JLM, TPROF etc, are available under this performance inspector to profile java application health. However, JLM is efficient enough to detect any contention bottlenecks in a java application. In order to detect contention related performance issue, performance engineers usually follow some manual steps while using these tools. These steps are:

1. **Observe Perf data:** Let's assume, our example code `synchronized task` has a performance issue with comparatively low throughput. As a performance engineer, it is recommended to perform the perf testing before analyzing any other tools. However, the perf tool records the kernel's memory footprint and collects the samples of the symbols printed on the memories. The sweet spot of the perf recording is, it collects all the symbol names that are either from operating system's tokens or tokens used in the user space applications such as java application. Moreover, in case of any issues with the application perf data captures different signatures. Hence, if the application encounters with contention-related issue then those related symbols will be reflected in the perf data.

After scanning through the perf data, performance engineers capture the most probable hottest region of the application due to heavy contention along with the contention-related symbols. A single snap-shot of perf data for our example code "SyncTask" is shown in Figure 2.1. However, if we look carefully then we can see that some symbols related to contention are marked with red lines. Additionally it is also visible that the method "run" from the class "SyncTaskThread" is reflected on the perf trace which is the hottest region of the example code.

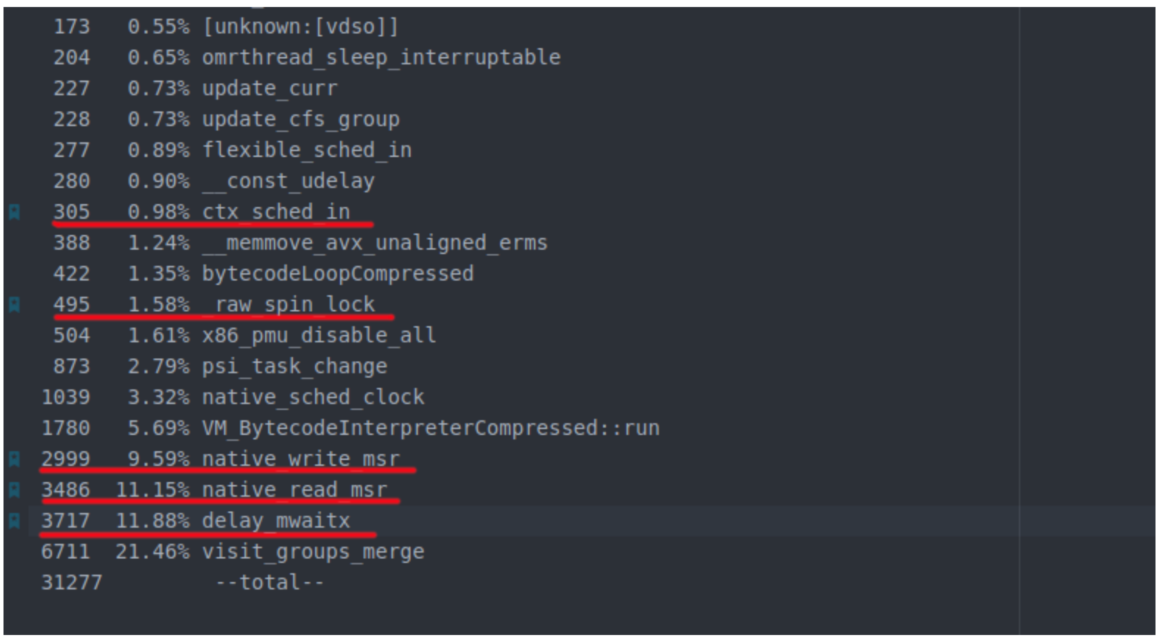


Figure 2.1: A single perf snapshot for Sync Task example code indicating high sample counts for some contention-related symbols

2. **Observe JLM data:** Now, it has been validated with probability that the issue is contention-related bottlenecks, it is worth looking at the JLM data next. As a performance engineer, it is then recommended to run and activate the JLM to collect statistical information related to highly contended monitors from its agent. JLM collects contention-related statistics using the agent, and this agent should be included as a run-time argument while running the java application. After capturing the JLM data, performance engineers typically scan through the

"Java Inflated Monitors" block to obtain a high-level overview of contented monitors and the statistics. A single snapshot of JLM data for the "SyncTask" example code is shown in Figure 2.2. Perf log reveals the hottest region; in contrast, JLM exposes the monitors responsible for high contention and possible reason behind the hottest part of the code.

```
      0        0        0        0        0        0        0     0        0  [00007F70280D7680] MM_FreeEntrySi
      0        0        0        0        0        0        0     0        0  [00007F70280D7828] MM_FreeEntrySi
      0        0        0        0        0        0        0     0        0  [00007F70280D7300] MM_MemoryPoolA
      0        0        0        0        0        0        0     0        0  [00007F70280D7198] MM_MemoryPoolA
      0        0        0        0        0        0        0     0        0  [00007F7028056CD8] MM_FreeEntrySi
      0        0        0        0        0        0        0     0        0  [00007F70280DE280] MM_FreeEntrySi
      0        0        0        0        0        0        0     0        0  [00007F70280DE428] MM_FreeEntrySi
      0        0        0        0        0        0        0     0        0  [00007F70280E38D0] MM_MemorySubSp

  Java (Inflated) Monitors

  %MISS     GETS   NONREC     SLOW      REC    TIER2    TIER3 %UTIL AVER_HTM  MON-NAME

    100       18       18       18        0        0        0    96 2016737495  [00007F6F880027A8] SyncTask2@00

  LEGEND:
                 %MISS : 100 * SLOW / NONREC
                  GETS : Lock Entries
                NONREC : Non Recursive Gets
                  SLOW : Non Recursives that Block
                   REC : Recursive Gets
                 TIER2 : SMP: Total try-enter spin loop cnt (middle for 3 tier)
                 TIER3 : SMP: Total yield spin loop cnt (outer for 3 tier)
                 %UTIL : 100 * Hold-Time / Total-Time
               AVER-HT : Hold-Time / NONREC
```

Figure 2.2: A single JLM snapshot for Sync Task example indicating contention due to high hold time reflected in key AVER_HTM

After obtaining the JLM data, engineers usually perform the analysis, such as if the AVER_HTM count is high, then it is assumed that the locking issue is related to fault type 1 where the threads are holding the lock more than expected. Based on this observation, engineers usually return to the code base and try to reduce contention by removing excessive work under the critical section. However, the suggestion is made for holding the locks for long, but suggestions are unavailable for fault due to increased access to the locked resources by threads with high frequency.

3. **Locate Bottleneck Area:** At this stage, performance engineers have the JLM contention results and symbol names possibly responsible for the hottest regions of the application. They usually move to the next step, where they dig deeper into the call stacks and search for those symbols (method names). However, after expanding the call stacks, engineers point out the code blocks responsible for poorly managed concurrent code due to inefficiently managed locks.

2.3.2 YourKit Java Profiler

YourKit [9] is a popular java profiling tool, commercial purpose, closed-source software built by YourKit GmbH. This profiler is capable of capturing java applications' profile data and is widely used by performance engineers to monitor the java applications' health. In order to capture profile data, YourKit profiler utilizes its agent tool, which needs to be prepared prior to the execution of the java application. Besides some other extensive features, enabling one provides the opportunity to capture the profile data both for java application / JVM running in a local environment or a remote machine. Similar to other java profiling tools, it is also capable of seizing the data for CPU usage, Memory usage, Threads & Monitor activities, etc. YourKit profiler comes with a robust graphical user interface that provides the most manageable navigation features to the user. Users are allowed to pause, resume and stop capturing profile data and events running in a VM once the profiler starts. As a performance engineer, one has to follow the steps below to collect the profiling data using this tool:

1. **Prepare Agent:** Before moving forward, a performance engineer should prepare the agent library for the application to be attached as a run-time argument. Several options can be configured for the agent, such as CPU profiling, threads, and monitors profiling, exceptions profiling, memory usage profiling, etc. In order to enable the agent to start collecting profile data, an engineer has to select an option from the home window titled "Profile local or remote java applications". The home window of the YourKit profiler is shown in Figure 2.3. One can

find the listed JVMs running in a local or remote machine under the "Monitor Applications" section of the YourKit application UI. Once the agent configuration is done and attached to the java application as a run-time argument, YourKit java profiler finds the underlying java pids. It lists them under this section to allow users to start profiling. See Figure 2.4 that shows the area engineers should navigate to begin profiling.

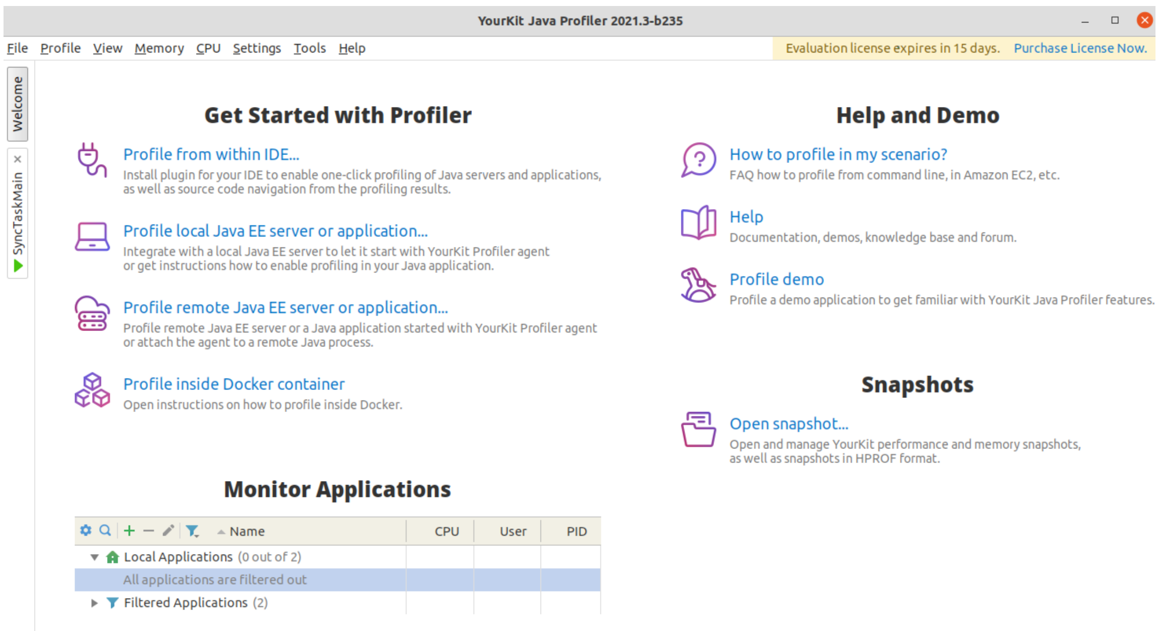


Figure 2.3: Home window of YourKit java profiler allows user to start profiling applications

2. **Capture Profile Data:** After configuring the necessary arguments, the agent is now ready to collect the profiling data. Engineers attach the agent as a run-time argument to the application, and it starts displaying the data to the YourKit application UI. However, there are options in the YourKit application UI to capture the data for specific profiling whenever it is needed. For our example application "SyncTask", thread activities profiling and monitor usage profiling are captured. The thread and the monitor profiling are shown in Figure 2.5 and Figure 2.6 respectively.

3. **Understanding Contention:** Figure 2.5 demonstrated the profiler's action that illustrates

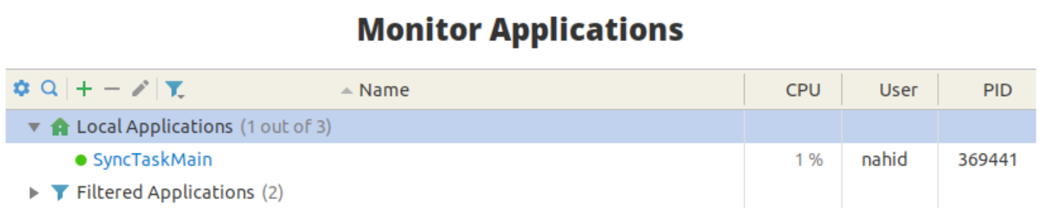


Figure 2.4: Available applications list window of YourKit java profiler allows user to start profiling applications

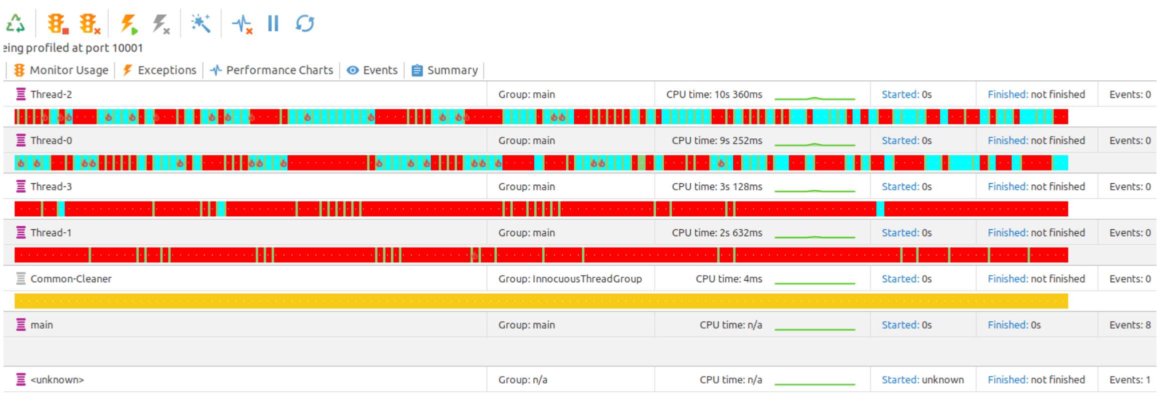


Figure 2.5: Thread activities profiling using YourKit java profiler for SyncTask example

the thread progression created by our example concurrent program. As the example runs with only four threads, the figure pictures four thread-progress bars with necessary colors indicating numerous states of the thread activities. However, the progress bars contain two different colors, red and green. The green color indicates the active or running state of the thread, and the red time-frame for the thread progress bar reflects the blocking state of the thread. When any thread(s) are blocked by other thread(s) more than the usual time, that situation can be concluded as a sign of contention. In the thread activity bars, it is visible that the percentage of red color is way more than the green color, indicating contention.

After analyzing the thread activities, it is required to know the contended monitors in our application. The "Monitor Usage" tab of the profiler shown in Figure 2.6 illustrates the high-level overview of monitor uses and the waiting or blocked states of different threads. The corresponding thread for which the other thread is blocked is also noticeable from the monitor usage window of YourKit Profiler.

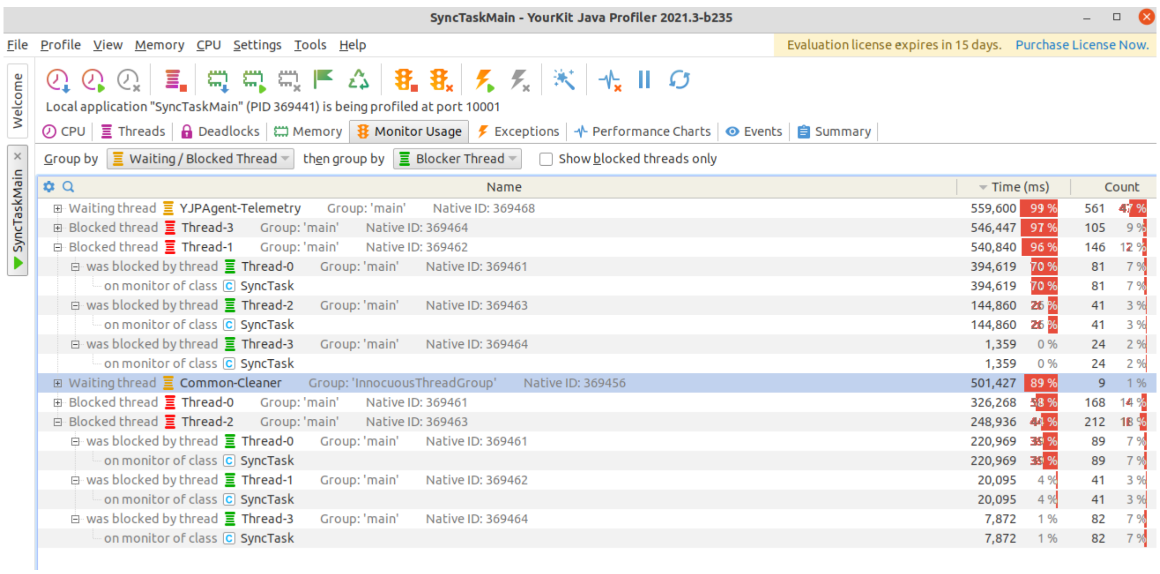


Figure 2.6: Monitor usage profiling for SyncTask example using YourKit java profiler

2.3.3 JProfiler

JProfiler [10] is a closed-source and commercially licensed java profiling tool available in the market developed by ej-technologies GmbH, targeted at Java EE and Java SE applications. In order to analyze and visualize the lock-related performance issue using JProfiler, performance engineers usually operate the application graphical user interface. It comes with a powerful graphical user interface that let us perform profiling an application with ease. Similar to other common profiler it provides profiling options to perform analysis for local application as well as applications running in a remote machine. Leveraging dynamically-linked shared library, it enables connecting JVM in either local machine or remote and start profiling and collecting data. It also provides a headless mode that is capable of profiling the application in silent mode and captures necessary logs, then stores them to a desired directory from where one can collect and proceed with the further analysis. Unlike YourKit profiler, this profiler does not require us to prepare agent for it and all the operations are performed through its UI. Another great feature of the JProfiler is of operable as plugin for eclipse development IDE. It enables both memory profiling to assess memory usage and dynamic allocation leaks and CPU profiling to assess thread conflicts. Following are the steps a performance engineer has to perform to profile a java application with contention-related bottlenecks

1. **Initialize New Session:** As a performance engineer one has to start the application executable and open the window where it is possible to initialize the new session or start the session that has been created before. New session refers to the process of collecting logs from beginning discarding the older sessions. Additionally, the “Attach” option in UI provides the opportunity to connect the profiler directly to a running VM. However, in order to start collecting the profiling data performance engineers have to choose any of these. In case of new session, one has to provide the necessary arguments for the profiler to be started, such as directory of the class or jar file, then command line arguments needed for the java application. And in case of “Attach” option, choosing running VMs is available in the UI. A snapshot of new session window of the JProfiler is shown in Figure 2.7.

2. **Observe the Thread Activities:** After starting the session or connecting to a running VM, JProfiler usually starts collecting the profile data. Different modules present different profile statistics such as the “Live Memory” window describes memory usage, “CPU View” enables profiling CPU usage, etc. However, performance engineers turn on the window of “Threads” and “Monitors & Locks” to visualize the contention-related performance. For demonstration purpose, we started our new session with a high number of sleep times and four threads that emulates the contention. Figure 2.8 demonstrates the thread progression bars similar to the YourKit java profiler. The threads are running with red colors, indicating different states

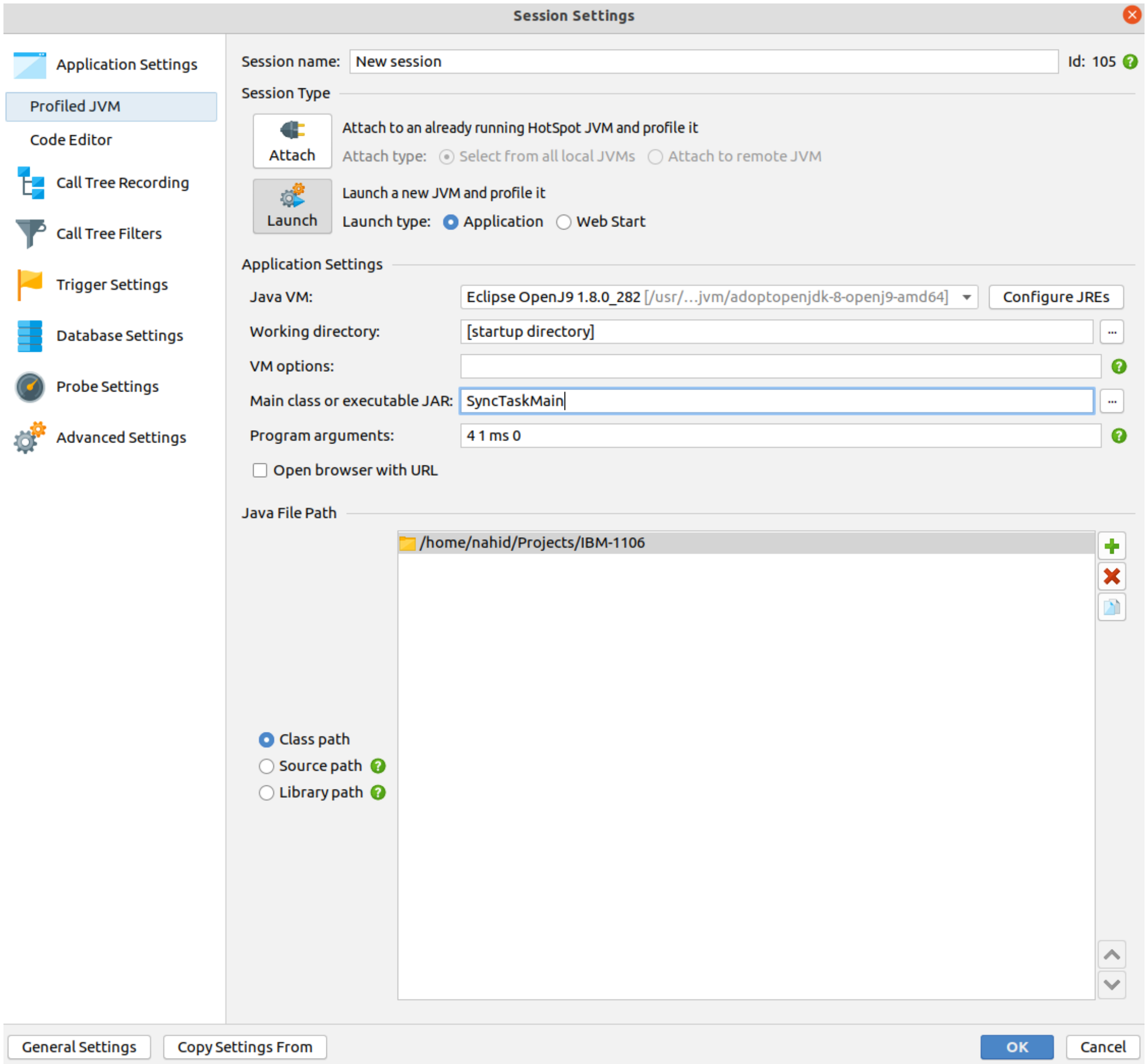


Figure 2.7: A snapshot of JProfiler's new session configuration window

of the threads' activities. Compared to the red colors, the percentage of the green color is deficient. Also, the yellow color represents the threads' waiting mode, which is also great in number.

3. **Observe the Monitors:** Observing poor conditions of the thread activities lead us to inspect the monitors of the application. Performance engineers need to know the monitors that are causing performance issues is listed under "Monitor Statistics" window of the JProfiler. For

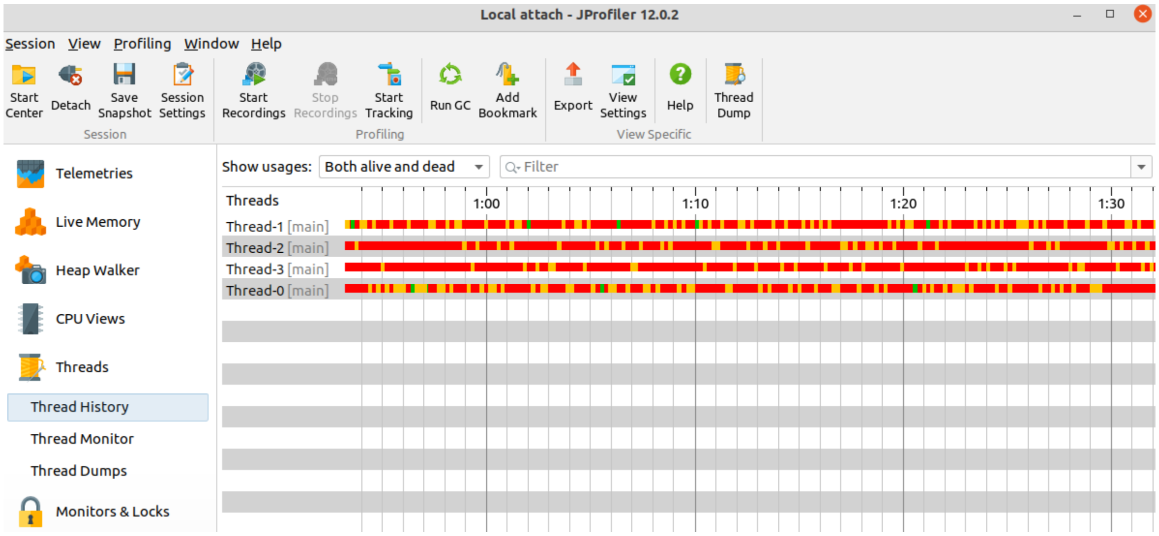


Figure 2.8: A snapshot of JProfiler's thread activities window captured for our SyncTask example code

our example SyncTask concurrent code the contended monitors statistics are listed under the monitor history, see Figure 2.9, and the monitor statistics, see Figure 2.10 respectively.

2.3.4 Visual VM

VisualVM [16] is an open-source tool that provides a visual interface for viewing java applications' performance while they are running on a Java Virtual Machine (JVM) and for troubleshooting problems and profiling them. It has lightweight profiling capabilities designed for both development and production time use. Java application developers can use Java VisualVM to troubleshoot applications and monitor and improve the applications' performance. Java VisualVM can allow developers to generate and analyze heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweight memory and CPU profiling. Developed by Oracle, this profiler was integrated with NetBeans IDE and comes as a default performance analyzing tool for java applications. Recently, NetBeans IDE discontinued the idea of integrating Visual VM as a default performance tool. Unlike YourKit profiler, this profiler also does not require us to prepare the agent

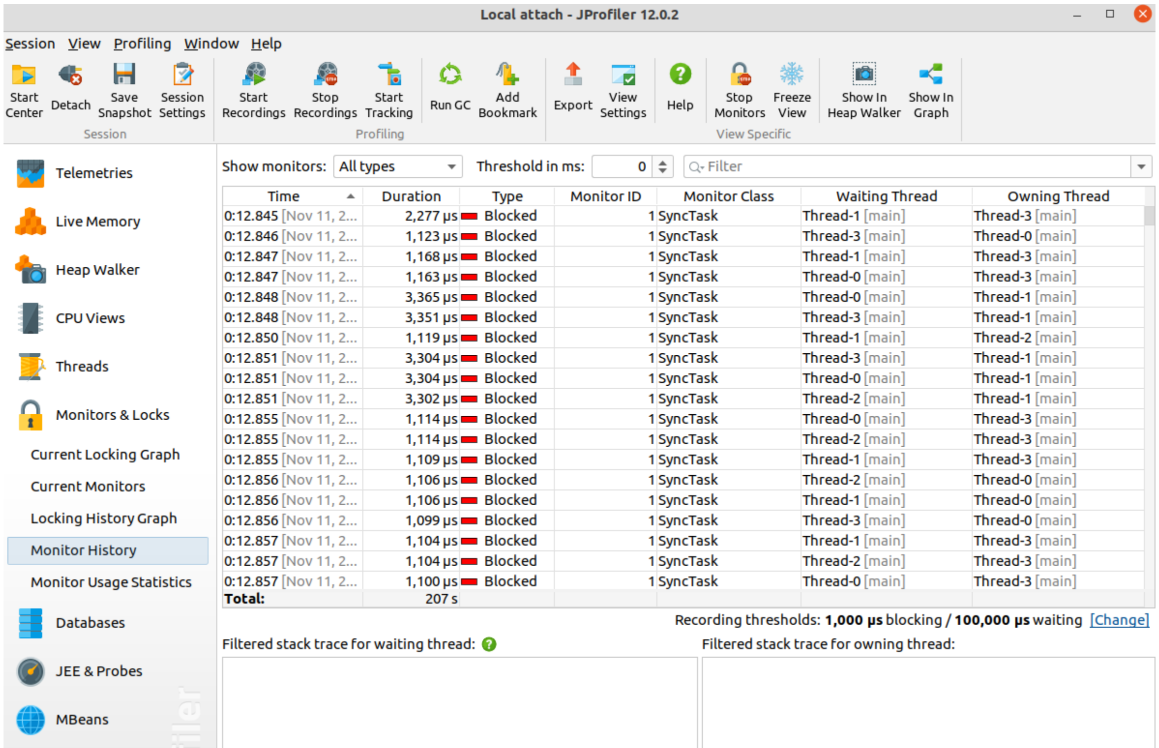


Figure 2.9: A snapshot of JProfiler's monitor history window captured for our SyncTask example code

for it. Instead, it captures the java application automatically running on a local or a remote machine. In the UI, it has a applications list bar where it shows all the captured running java applications. As a performance engineer, one should click on a particular application to start monitoring and profiling.

Monitoring the thread activities requires one performance engineer to move to the "Threads" tab of the UI window. Under this window, one can observe the real-time thread activities running in a java application.

A single snapshot of Visual VM is shown in Figure 2.11 where it presents the thread profiling of our example code "SyncTask" with contention. Thread numbers 0 to 4 are spotted that are being used in our "SyncTask" example code we are interested in. If we look carefully then it is clearly visible that the running time (green blocks) of 4 threads are noticeably low. Most of the time, they are blocked by each other. Moreover, it is also visible that the percentage of running time for those threads are

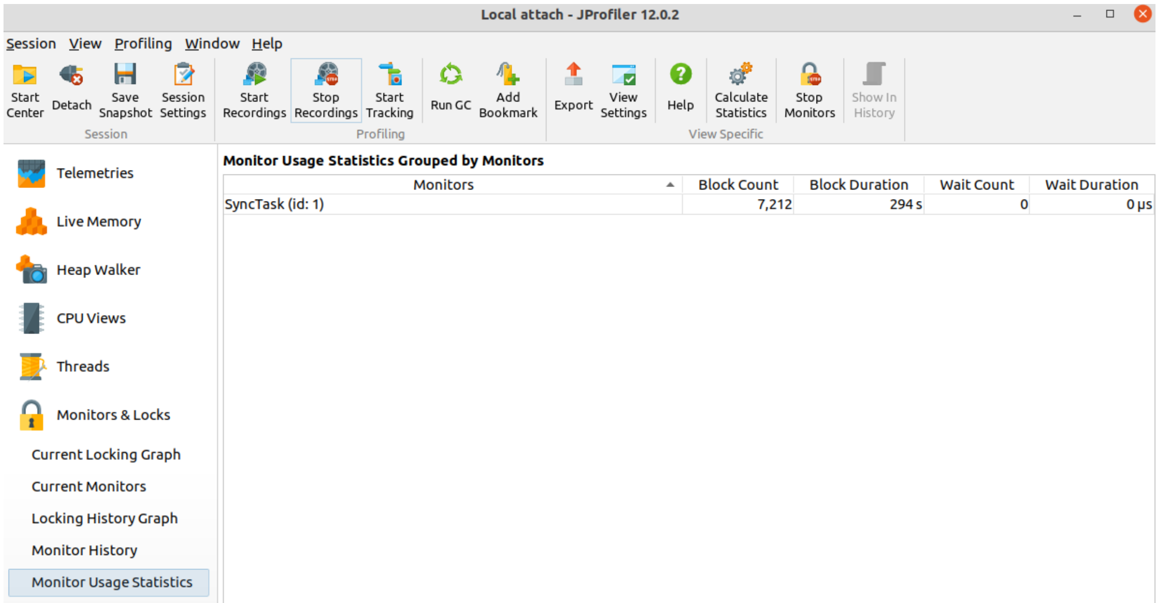


Figure 2.10: A snapshot of JProfiler's monitor statistics window captured for our SyncTask example code

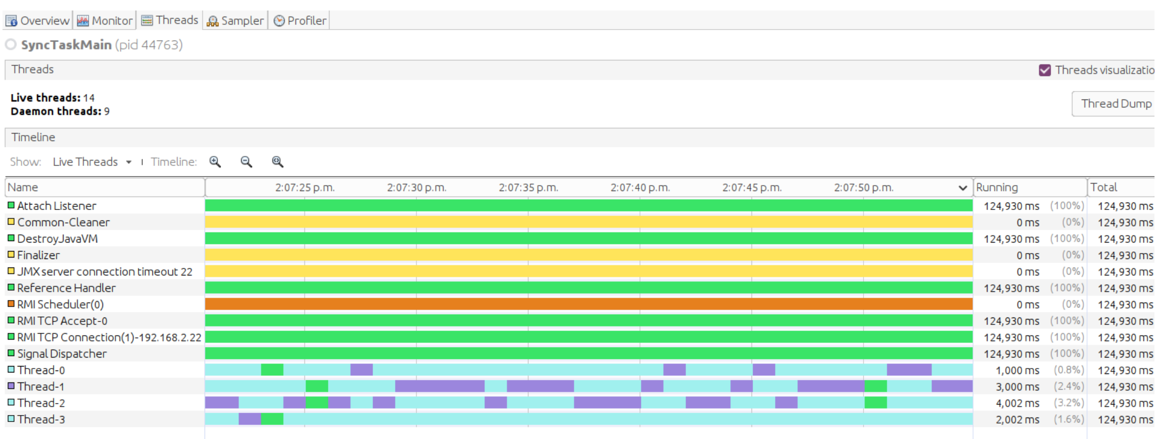


Figure 2.11: Visual VM thread profiling for sync task example

0.8%, 2.4%, 3.2% and 1.6% respectively. Additionally, Visual VM provides the option to take the total snapshot of the thread dump. A snapshot of the thread dump of our "SyncTask" example code is shown in Figure 2.12. Thread activities in the thread dump show that thread-1, thread-2, and thread-3 are blocked and waiting to lock the monitor object. And the thread-4 has gone timed-wait, which means it is sleeping at the particular moment when the thread dump is taken.

```
SyncTaskMain (pid 53973)
Thread Dump

"Thread-0" #12 prio=5 os_prio=0 cpu=481.84ms elapsed=26.76s tid=0x00007fa5103c0800 nid=0xd2e6 waiting for monitor entry
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - waiting to lock <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

   Locked ownable synchronizers:
        - None

"Thread-1" #13 prio=5 os_prio=0 cpu=886.98ms elapsed=26.76s tid=0x00007fa5103c2800 nid=0xd2e7 waiting for monitor entry
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - waiting to lock <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

   Locked ownable synchronizers:
        - None

"Thread-2" #14 prio=5 os_prio=0 cpu=448.61ms elapsed=26.76s tid=0x00007fa5103c4000 nid=0xd2e8 waiting for monitor entry
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - waiting to lock <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

   Locked ownable synchronizers:
        - None

"Thread-3" #15 prio=5 os_prio=0 cpu=920.40ms elapsed=26.76s tid=0x00007fa5103c6000 nid=0xd2e9 sleeping [0x00007fa4d2ff50
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(java.base@11.0.11/Native Method)
        at java.lang.Thread.sleep(java.base@11.0.11/Thread.java:334)
        at SyncTask.taskOne(SyncTaskMain.java:35)
        - locked <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

   Locked ownable synchronizers:
        - None

"DestroyJavaVM" #16 prio=5 os_prio=0 cpu=113.30ms elapsed=26.76s tid=0x00007fa510028000 nid=0xd2d6 waiting on condition
   java.lang.Thread.State: RUNNABLE
```

Figure 2.12: Visual VM thread dump for sync task example

2.3.5 JDK Utilities

JDK Utilities are mainly command line tools that are handy and provide quicker solution to analyze the thread activities and lock contention by taking a thread dump. Most of the JDK utility tools are available in bin directory under JDK home path. However, thread dump is a snapshot of the state of all the threads in a java process and it is written in plain text. Moreover, the thread dump contains the stack trace of the thread activities that allows performance engineers diagnose the locking-related problems with ease. There are several JDK Utilities available such as `jstack`, `jconsole`, `jcmd`, `kill` etc.

- **jstack:** jstack is operated in command line and requires java process id to capture necessary thread dump. The following command and options are used for jstack.
  jstack [-f][-l][-m] <java_process_pid>
  These -f/l/m flags are optional and have different uses. To capture the dump we can use the following:
  $ jstack -l <java_process_pid>
  It is also possible to redirect the output dump to a file and that requires the following final jstack command:
  $ jstack -l <java_process_pid> > jstack.out

- **kill:** Unix command kill with signal -3 is used to capture the java applications' thread dump. It dumps the output directly to the default java output if any logger is specified. However, it is also possible to redirect the dumps to a separate file which needs adding some run-time arguments before running the application. This kill command also requires java process id and that can be found using `ps aux` command in Unix-like systems. In order to send the kill signal we need to simply follow the command below:
  $ kill -3 <java_process_pid>
  In case of redirecting output to a separate file it is required to adjust some java run-time arguments:
  $ java -XX:+UnlockDiagnosticVMOptions -XX:+LogVMOutput -XX:LogFile=./dump.log Program.java

A snapshot of thread dump is taken using the kill command providing the above arguments and shown in Figure2.13. However, from the figure it is visible that except thread-2 the all other threads are blocked and waiting to acquire the lock at that particular moment. As a performance engineer, one should analyze these thread activities and based on this analysis he/she should conclude whether the current condition of the particular piece of code is well

```
"Common-Cleaner" #11 daemon prio=8 os_prio=0 cpu=0.11ms elapsed=19.10s tid=0x00007f4800353000 nid=0xdb57 in Object.wait()  [0x00007f47d036f000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(java.base@11.0.11/Native Method)
        - waiting on <0x000000062c389190> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(java.base@11.0.11/ReferenceQueue.java:155)
        - waiting to re-lock in wait() <0x000000062c389190> (a java.lang.ref.ReferenceQueue$Lock)
        at jdk.internal.ref.CleanerImpl.run(java.base@11.0.11/CleanerImpl.java:148)
        at java.lang.Thread.run(java.base@11.0.11/Thread.java:829)
        at jdk.internal.misc.InnocuousThread.run(java.base@11.0.11/InnocuousThread.java:134)

"Thread-0" #12 prio=5 os_prio=0 cpu=357.20ms elapsed=19.07s tid=0x00007f48003d1000 nid=0xdb58 waiting for monitor entry  [0x00007f47d026e000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - waiting to lock <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

"Thread-1" #13 prio=5 os_prio=0 cpu=336.29ms elapsed=19.07s tid=0x00007f48003d2800 nid=0xdb59 waiting for monitor entry  [0x00007f47d016d000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - waiting to lock <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

"Thread-2" #14 prio=5 os_prio=0 cpu=709.82ms elapsed=19.07s tid=0x00007f48003d4800 nid=0xdb5a runnable  [0x00007f4793ffd000]
   java.lang.Thread.State: RUNNABLE
        at SyncTaskThread.run(SyncTaskMain.java:131)

"Thread-3" #15 prio=5 os_prio=0 cpu=678.81ms elapsed=19.07s tid=0x00007f48003d6000 nid=0xdb5b waiting for monitor entry  [0x00007f4793efd000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at SyncTask.taskOne(SyncTaskMain.java:34)
        - locked <0x000000062c01e888> (a SyncTask)
        at SyncTaskThread.run(SyncTaskMain.java:131)

"DestroyJavaVM" #16 prio=5 os_prio=0 cpu=103.44ms elapsed=19.07s tid=0x00007f4800029000 nid=0xdb47 waiting on condition  [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"VM Thread" os_prio=0 cpu=9.62ms elapsed=19.15s tid=0x00007f4800264800 nid=0xdb4e runnable

"GC Thread#0" os_prio=0 cpu=2.88ms elapsed=19.17s tid=0x00007f4800041000 nid=0xdb49 runnable

"G1 Main Marker" os_prio=0 cpu=0.44ms elapsed=19.17s tid=0x00007f480007b800 nid=0xdb4a runnable

"G1 Conc#0" os_prio=0 cpu=0.08ms elapsed=19.17s tid=0x00007f480007d000 nid=0xdb4b runnable

"G1 Refine#0" os_prio=0 cpu=0.39ms elapsed=19.16s tid=0x00007f48001ff000 nid=0xdb4c runnable

"G1 Young RemSet Sampling" os_prio=0 cpu=3.37ms elapsed=19.16s tid=0x00007f4800200800 nid=0xdb4d runnable
"VM Periodic Task Thread" os_prio=0 cpu=15.22ms elapsed=19.11s tid=0x00007f4800344800 nid=0xdb56 waiting on condition
```

Figure 2.13: A snapshot of thread dump for sync task example taken using kill -3 command

performing or has a severe bottleneck.

2.4 LIMITATIONS OF TRADITIONAL APPROACHES

The tools listed above and similar approaches have been utilized for more than a decade, helping developers detect bottlenecks and bugs efficiently. As the java language runs on a virtual machine, and sometimes bottlenecks occur due to VM issues, these tools typically come with the java language itself or are integrated with IDE. Thus, these tools were essential from the beginning. However, these tools have some limitations; we can list them below:

1. They need human intervention to debug the problematic situations and locate the places.

2. They are unable to suggest a proper recommendation as they are incapable of analyzing the profile data.

Although these tools are efficient, the recent need of the developers and based on the listed above limitations motivated us to conduct this research as to whether it is possible to throw some proper recommendations along with reducing manual human interventions detecting the problems. For instance, we ran our example code "SyncTask" with a contention issue. Our example code can be emulated to create both of the two issues a) Type-1: High hold time, b) Type-2: High frequent access requests. However, we emulated with 4 threads and applied 1 millisecond inside the critical section. In this case, the program experienced a high hold time contention. The tools listed above can only detect the threads are taking too much time and are responsible for contention, but the analysis fails to describe the type of contention bottleneck it is experiences. More precisely, if the contention would happen due to high frequent access request fault, these tools fail to conclude the proper reasons and statement.

2.5 Summary

This chapter summarizes some related works and classical ways of debugging contention bottlenecks. In the related works section, we try to present some approaches around this area, solving contention bottlenecks by analyzing the critical section pressure, unnecessary loops around the critical section, and many more. These are good approaches and only deal with contention due to spending extra time at the critical section but failing to distinguish the root issues described in Goetz's book. Moreover, these approaches lack analyzing the performance data that can be useful to detect contention bottleneck types.

Tools presented at the end of the chapter are efficient enough but limited to describing bottleneck types.

Chapter 3

METHODOLOGY

3.1 INTRODUCTION

The hypothesis that drives our methodology is that a lock can experience performance bottlenecks by producing some amount of contentions under any circumstances. However, contentions can be accelerated by either having some operations that hold the lock more than expected or access the lock with high frequency. Based on this hypothesis, we are interested in determining if lock contention faults can be classified into the two potential causes described by Goetz [3]:

- Type 1 - Threads spend too much time inside the critical sections, and
- Type 2 - High frequency with which threads access the critical section.

Although we are interested in two classes referring to the two potential faults, our generated data should contain more categories than the two. During execution of our example code and generating data, we varied from low number of threads to high number of threads and low sleep time to high sleep time. In this chapter, we try to detail our methodology in several steps that are needed to complete our approach. Steps such as acquiring run-time metrics, then filtering and aggregating the metrics into single file and lastly data preprocessing and classification are part of our approach. These steps are shown in a high-level workflow (see Figure 3.1), where we show the data flow from exercising the example code till the classification process. In the run-time metrics acquisition step, we show the process we go through to acquire run-time metrics from exercising the example code. The filtering and aggregation step describes the process where we collected data from different sources and merged them into a single file. It is ideal that the data we fed into an ML model should be streamed from a single source rather than multiple files. And in our last preprocessing and classification step, we show the different phases we followed by applying some data preprocessing algorithms along with some unsupervised fashioned clustering techniques to classify the fault types.

3.2 Approach

As a preliminary approach, our methodology uses several run-time logs from a Linux *perf*, and JLM performance analyzers, then analyze them using a KMeans classifier to determine the existence of different types of lock contention faults. Before analyzing the data using pure KMeans, we preprocess our data, scale, and reduce the features leveraging Principal Component Analysis (PCA). Instead of feeding the raw data to KMeans, this processed data is more important to the algorithm. Typically, clustering algorithms understand the processed and fine-tuned data more than the unprocessed raw data. Therefore, that processing help find the actual clusters out of the dataset. We also call our methodology a preliminary approach because there are no available datasets for this research. First, we collect performance metrics data and generate a dataset by running some concurrent codes that create contention. Then we analyze the dataset utilizing KMeans to understand the insight of the data.

Analyzing the data using KMeans and finding the expected classes falls into the unsupervised clustering technique. As a result, our methodology has the portion where an unsupervised learning technique is introduced to classify the contention types. Before moving forward, a discussion regarding incorporating unsupervised learning is necessary to the readers. This research work is based on the performance metrics (e.g., GETS, AVER_HTM, _raw_spin_lock) that mainly come from the performance analyzer tool and do not contain the labels of contention fault types. Therefore, in order

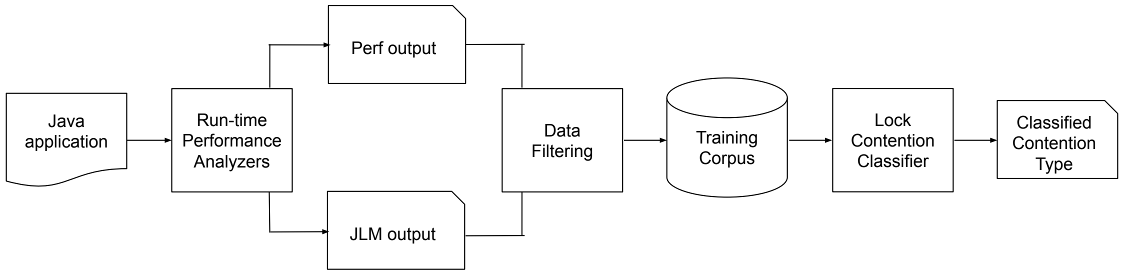


Figure 3.1: High level workflow of our approach

to classify the contention fault types, a labeled dataset is needed, which is unfortunately unavailable despite our searching efforts. Hence, the methodology introduces the generation of a dataset, clustering them, and labeling them, which lead the whole ML approach to an unsupervised learning technique.

Our expected unsupervised classification process depends on several steps that are listed below and further detailed in the following sub-sections and shown in figure 3.1.

1. Run-time performance metric acquisition;

2. Aggregation and filtering of the metrics from the logs;

3. Data pre-processing and Classification.

3.3 Method Steps

3.3.1 Run-time performance metric acquisition

A Java exemplary code emulating lock contention is executed in a controlled environment leveraging the *perf* and JLM tools that result in particular performance metrics. The code is executed using these performance tools multiple times to reduce the effects of outliers in the metrics and we usually skip the first $10s$ of the execution to avoid the JVM's code optimization and warm-up period. In order to cover a variety of contention scenarios we vary the time that a lock is held by the application as well as the number of threads that use the lock.

Our run-time performance metric acquisition step mainly depends on two performance analyzer tools named *perf* and Java Lock Monitor (JLM).

1. **JLM:** JLM stands for Java Lock Monitor that was previously built for and a part of IBM Performance Inspector [8] tool-suite to diagnose Java application's health. JLM is capable enough to capture the contention statistics when an application experience some level of contentions. However, a profiling agent associated with JLM called "JPROF" mainly added to the run-time argument list prior to running a java application. This agent tool helps capture the information about lock usage for JLM from a running java application among several logs. JLM data mainly contains the two statistics related to monitors used by the operating system and the java program or the JVM itself. The two statistics are elaborately listed under labels "System (Registered) Monitors" & "Java (Inflated) Monitors" respectively. These two blocks of data are important to any performance analyst because these data describe the overall contention statistics that a java application experiences at that moment. However, our primary focus remains on the contention statistics related to Java monitors only which is required for our classification.

   Although, JLM provides quite a few metrics related to java inflated monitors but these are not well defined or documented. In order to move forward with these metrics and make ourselves familiar with them better the reader must know the details about them. The details are provided below:

   - %MISS: Application locks failure percentage
   - GETS: Total number of applications lock = the FAST + SLOW + REC
   - NONREC: Total number of non recursive application lock (Non Recursive GETS)
   - SLOW: Non recursive that block

- REC: Recursive GETS (thread requests for lock and already acquired it)
- TIER2: On platforms that support 3-layer spin locks, the number of inner loops to obtain locks.
- TIER3: On a platform that supports 3-layer spin locks, the number of cycles in the outer layer to obtain the lock.
- %UTIL : 100 * Hold-Time / Total-Time
- AVER-HTM : Hold-Time / NONREC

When contention occurs, JLM lists the java monitors used in our code under the "JLM Inflated Monitors" with some high counts for each of its metrics. A glimpse of the JLM log is shown in Figure 3.2. When contention occurs, and it has high counts in AVER_HTM compared to a low contention scenario shown in Figure 3.3. In case of no contention, the java lock monitors do not appear or often appear with all zero values for monitor columns in the JLM log under the "Java inflated monitor" block. During a low contention, the monitors appear with less count for monitor column such as AVER_HTM. Hence, contention due to hold time focuses on that specific AVER_HTM column. Heavy contention due to high hold time or comparatively low contention due to low hold time is not distinguishable easily with the bare eyes. Therefore, it is a key factor to why a classifier is essential as determining these thresholds is not straightforward. A glimpse of the JLM log for sync task example with less contention is shown in the figure 3.3.

2. ***perf*** The *perf* tool comes with the Linux distribution by default which is another essential tool has the equal contribution to our research as same as the JLM. The *perf* tool is capable of capturing memory footprints, in other words, symbols from user space and kernel space. These symbols are mainly method names, variables, or class names usually used in the OS itself or the kernel or in a java application. Additionally, *perf* aggregates the symbol's frequency that is useful to predict the fault types. The reference of how the *perf* tool works can

```
    0       0       0       0       0       0       0    0       0  [00007F70280D7680] MM_FreeEntrySi
    0       0       0       0       0       0       0    0       0  [00007F70280D7828] MM_FreeEntrySi
    0       0       0       0       0       0       0    0       0  [00007F70280D7300] MM_MemoryPoolA
    0       0       0       0       0       0       0    0       0  [00007F70280D7198] MM_MemoryPoolA
    0       0       0       0       0       0       0    0       0  [00007F7028056CD8] MM_FreeEntrySi
    0       0       0       0       0       0       0    0       0  [00007F70280DE280] MM_FreeEntrySi
    0       0       0       0       0       0       0    0       0  [00007F70280DE428] MM_FreeEntrySi
    0       0       0       0       0       0       0    0       0  [00007F70280E38D0] MM_MemorySubSp

Java (Inflated) Monitors

%MISS     GETS   NONREC     SLOW      REC    TIER2    TIER3 %UTIL  AVER_HTM  MON-NAME

  100       18       18       18        0        0        0    96 2016737495  [00007F6F880027A8] SyncTask2@00

LEGEND:
              %MISS : 100 * SLOW / NONREC
               GETS : Lock Entries
             NONREC : Non Recursive Gets
               SLOW : Non Recursives that Block
                REC : Recursive Gets
              TIER2 : SMP: Total try-enter spin loop cnt (middle for 3 tier)
              TIER3 : SMP: Total yield spin loop cnt (outer for 3 tier)
              %UTIL : 100 * Hold-Time / Total-Time
            AVER-HT : Hold-Time / NONREC
```

Figure 3.2: JLM output log of Sync Task example when contention occurs

be found here [11]. Unfortunately, the raw *perf* data is not human-readable. However, with the help of a script, we can extract a human-readable log containing the following 3 columns of values a) **Sample count**, b) **Percentage in total sample count**, c) **Symbol name**.

The perf command "perf-record" collects lots of symbols that are not related to contention faults. A group of symbols does appear when contention occurs. They usually appear with a high number of samples in case the code experience bad contention and with fewer samples when it experiences minimal contention on the other hand. After the execution of our example code multiple times, these symbols are well observed and taken to consideration for further processing. Leveraging these symbols might help us to identify the contention fault types, which is what we expect. In our dataset, the symbols represent the feature and the sample count as the value for the feature. A snapshot of the *perf* log of Sync Task example code is shown in figure 3.4 highlighting the some symbols such as **"_raw_spin_lock"**, **"delay_mwaitx"**, **"native_write_msr"** and **"native_read_msr"** that are

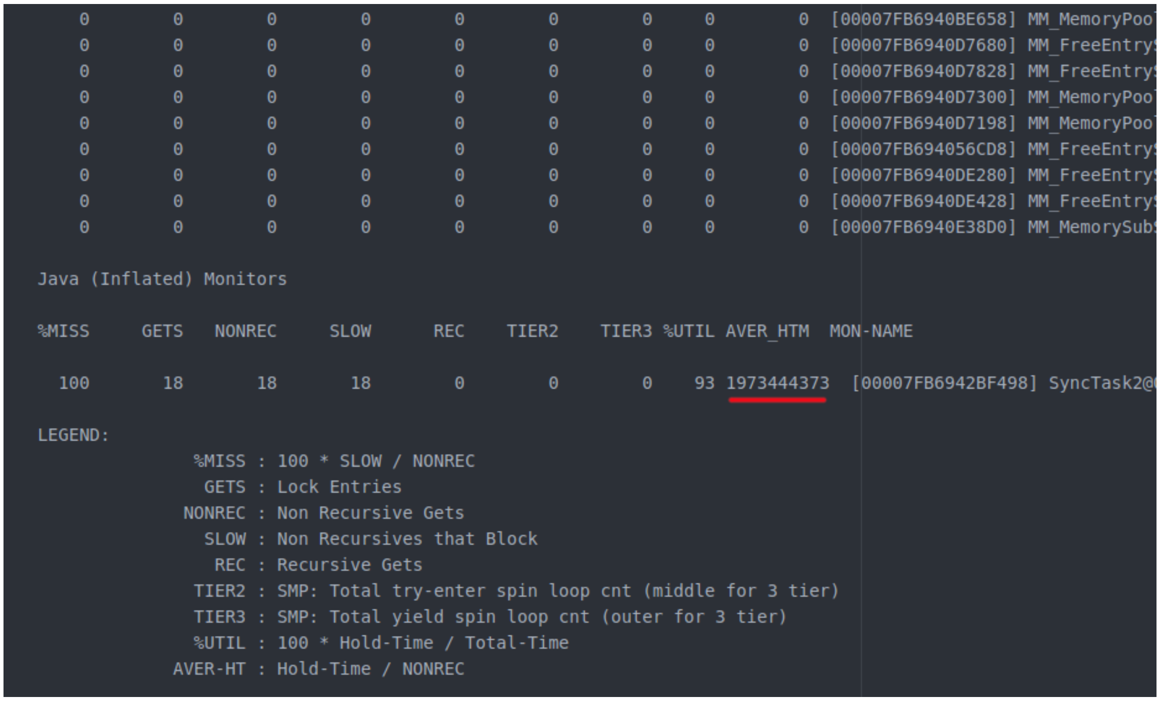
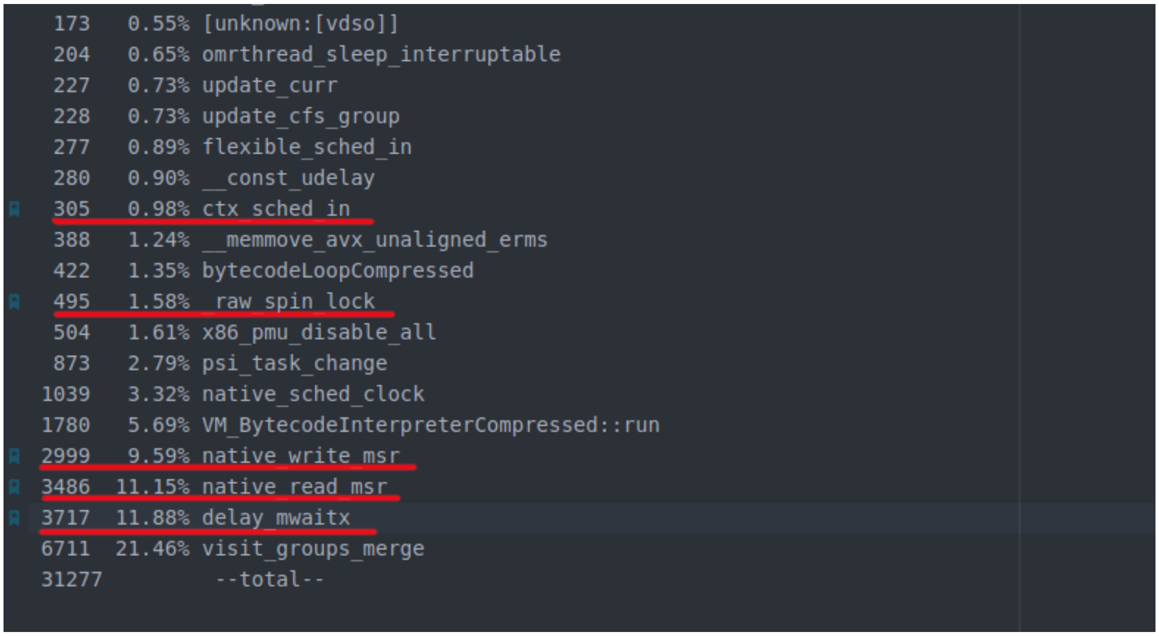


Figure 3.3: JLM output log of Sync Task example code when less contention occurs

consuming significant CPU resources compared to the others. However, in case of low contention these symbols often appear with a lower sample value. A snapshot of the *perf* log of Sync Task example code is shown in figure 3.5 when a low contention occurs.

In order to retrieve the performance metrics, it is required to run the exemplary concurrent code and extract the logs utilizing the performance analyzer tools. In a manual operation of log generation, we usually run the java application in one terminal. Next, opening a second terminal, we capture the application `pid`, which is required as an argument for the other shell commands of *perf* and JLM in order to collect further data. The command `ps aux | grep java` grabs the `pid` of the running java process, required as an argument for the perf command. As the Java `pid` is present, we start capturing *perf* trace using the command `perf record`, which extracts the “perf.data” file. However, the captured raw “perf.data” is not human-readable and requires a parsing operation. Therefore, the parsing is ensured utilizing a python script called “perf-hottest”



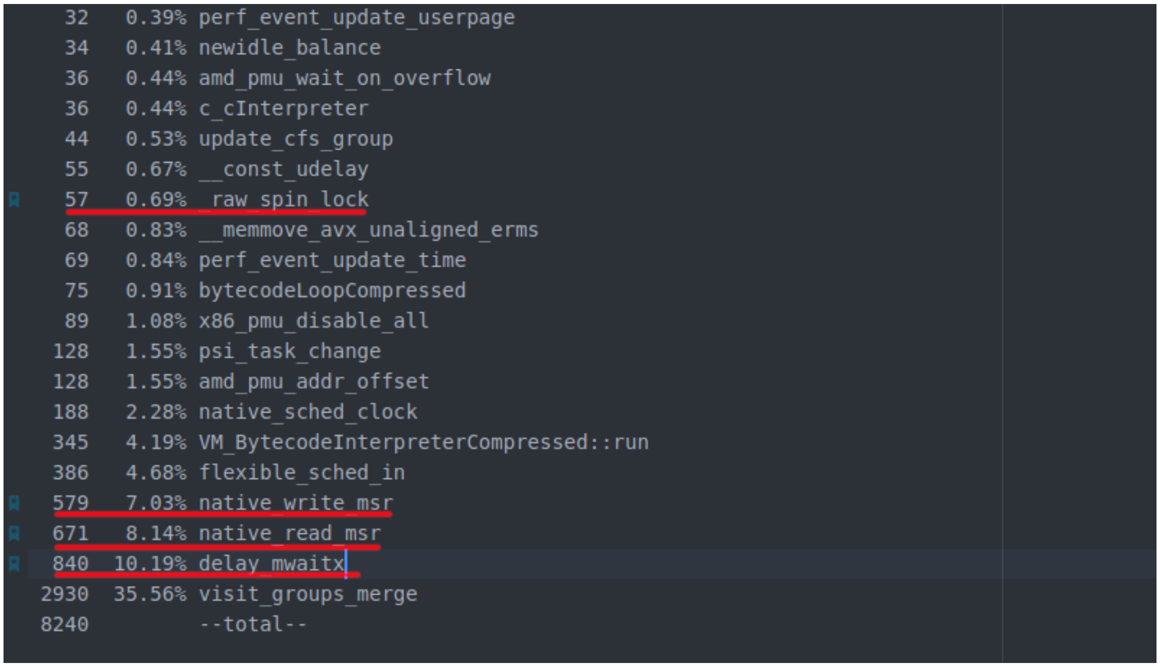
```
  173   0.55% [unknown:[vdso]]
  204   0.65% omrthread_sleep_interruptable
  227   0.73% update_curr
  228   0.73% update_cfs_group
  277   0.89% flexible_sched_in
  280   0.90% __const_udelay
  305   0.98% ctx_sched_in
  388   1.24% __memmove_avx_unaligned_erms
  422   1.35% bytecodeLoopCompressed
  495   1.58% _raw_spin_lock
  504   1.61% x86_pmu_disable_all
  873   2.79% psi_task_change
 1039   3.32% native_sched_clock
 1780   5.69% VM_BytecodeInterpreterCompressed::run
 2999   9.59% native_write_msr
 3486  11.15% native_read_msr
 3717  11.88% delay_mwaitx
 6711  21.46% visit_groups_merge
31277          --total--
```

Figure 3.4: A small portion of the perf snapshot taken for Sync Task code when contention occurs

that helps extract the human-readable perf information. Enabling all the perf commands require adding "-Xjit:perfTool" as a run-time argument during the execution of the java application.

Now, in order to obtain the log for JLM, we enable a different JVM run-time argument called **"agentlib:jprof"**. After running the java application with the necessary JVM options, we activate the **"rtdriver"** program to collect the JLM trace from another terminal. The **rtdriver** is also a part of the IBM performance inspector suite capable of seizing the JLM information from either a local machine or a remote one when the machine IP address is specified within the **rtdriver** command. This piece of software tool starts collecting the data sending the **"start"** signal and stops it when it sends **"stop"** command to the targeted machine. The detailed tools installation and log generation process is explained in Chapter 4.2.4.

Executing the code and producing the run-time perf and JLM log is a tedious and time consuming one. In order to accelerate the log generation process faster and then generating the dataset, we



```
  32   0.39% perf_event_update_userpage
  34   0.41% newidle_balance
  36   0.44% amd_pmu_wait_on_overflow
  36   0.44% c_cInterpreter
  44   0.53% update_cfs_group
  55   0.67% __const_udelay
  57   0.69% _raw_spin_lock
  68   0.83% __memmove_avx_unaligned_erms
  69   0.84% perf_event_update_time
  75   0.91% bytecodeLoopCompressed
  89   1.08% x86_pmu_disable_all
 128   1.55% psi_task_change
 128   1.55% amd_pmu_addr_offset
 188   2.28% native_sched_clock
 345   4.19% VM_BytecodeInterpreterCompressed::run
 386   4.68% flexible_sched_in
 579   7.03% native_write_msr
 671   8.14% native_read_msr
 840  10.19% delay_mwaitx
2930  35.56% visit_groups_merge
8240         --total--
```

Figure 3.5: A small portion of the perf snapshot taken for Sync Task code when less contention occurs

write an algorithm (steps listed below) capable of running the entire process multiple times. The algorithms that help to run the entire log generation process are shown in Algorithm 2 and Algorithm 1. The combination of the two algorithms are the automated steps for faster log generation, which is shown below:

- **Automated steps for faster log generation:**
  - Set thread number and sleep time
  - Run java program
  - Wait ten seconds
  - Execute *perf* and JLM for ten seconds
  - Terminate java program
  - Collect PERF and JLM data

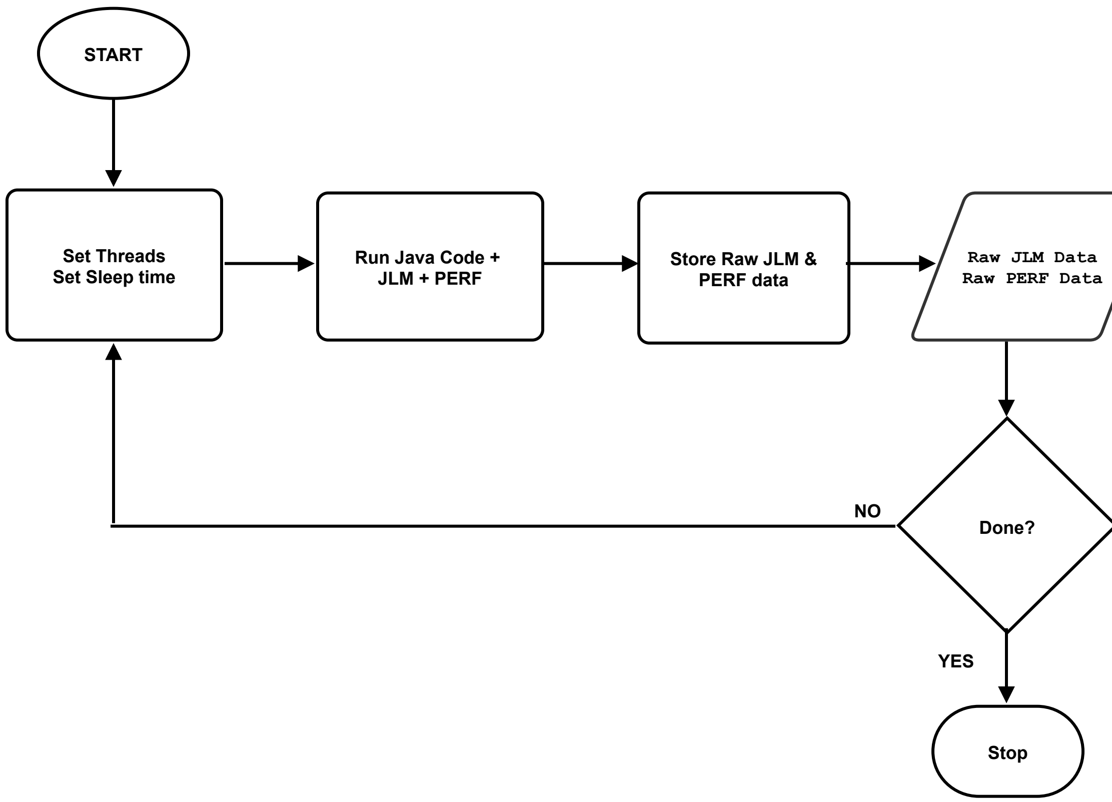


Figure 3.6: Run-time performance metric acquisition

– Repeat n times

It is required to run our code multiple times, varying the sleep times and thread numbers to emulate contention type 1 and contention type 2. The algorithm is shown in Algorithm 2 assist us in running a single test and collecting the necessary JLM and *perf* data, and saving them. Now, another algorithm script (see Algorithm 1) is needed to run our previous algorithm script (see Algorithm 2) multiple times to generate the whole dataset without human intervention. The algorithm for running the java codes and collecting the data multiple times is shown in Algorithm 1. This process not only makes the log generation effortless but also reduces the code exercise time. This algorithm consists of two loops, where the outer loop iterates over an array of thread numbers, and the inner loop is responsible for constructing a different sleep time compared to the previous run. Inside the

Algorithm 1: Algorithm to run collect data bash algorithm multiple times to collect JLM data and perf data and store them in a desired directory

```
THREADS ← initialize thread list;
for each THREADS do
    TRIALS ← number_loop;
    DELAY ← delay_value;
    for loop until TRIALS do
        execute Algorithm 2 to collect jlm and perf data;
        DELAY ← DELAY + 100;
    end
end
```

second loop, we run our data collection algorithm (see Algorithm 2) providing the necessary java application arguments such as thread numbers, sleep time, sleep time type. The sleep time type, in this case, emulates whether the java application should apply milliseconds to the critical section or nanoseconds.

Algorithm 2: Algorithm to run SyncTask example code, after that collect JLM data and perf data and store them in a desired directory

```
compile java program;
run java program with arguments [−Xjit:perfTool, −agentlib:jprof];
java_pid ← capture pid of java program using ps aux | grep | awk;
if java_pid not empty then
    record jlm data using rtdriver;
    sleep 10 seconds;
    record perf data;
    sleep 10 seconds;
    perf_pid ← capture perf pid using ps aux | grep | awk;
    kill perf pid using −SIGINT;
    kill java pid using −SIGKILL;
else
    notify java pid not found;
end
convert raw perf.data to perf.log using perf_hottest;
save jlm.log to desired dir;
save perf.log to desired dir;
```

Run-time performance metric acquisition process flow-chart is shown in Figure 3.6. Two of the processes **"Run Java Code + JLM + PERF"** and **"Store Raw JLM & PERF Data"** are the part of the algorithm 2, responsible for collecting and storing PERF and JLM data only.

3.3.2 Aggregation and Filtering

The dataset is run through a series of algorithms that merge different runs and data sources into one file for ease of access after collecting the data. Before merging the JLM data and *perf* data into a single one, a parser is needed to parse valuable information from these raw data. In order to achieve this, we write and prepare a parser algorithm using python language that parses the JLM data, *perf* data, and the test information into three different CSV files. The filtering and aggregation process is shown in Figure 3.7. While running a single test, we vary the threads and sleep times that are considered to be the test parameters in our case. These test parameters are needed in the future for evaluation and verification purposes during the clustering process. However, we store the JLM, *perf* and test information log containing a timestamp in their name to identify them as a single run. JLM data contains two main blocks are titled "System Registered Monitors" and "Java Inflated Monitors". As the java inflated monitors come from the user space such as the java applications, our interest is still stuck into these monitors that the JVM mainly uses for the locks. Moreover, these monitors contain the contention statistics which are needed for our work. An algorithm for parsing the JLM data is shown in the Algorithm 3.

We process the *perf* data separately as the perf data is collected from a different source. While this is underway the *perf* data will be filtered so that only the most significant symbols related to lock-contention are kept such as **"_raw_spin_lock"**, **"ctx_sched_in"**, **"delay_mwaitx"** etc. As the perf log has plenty of other symbols we are not interested, collecting only the symbols related to contention is a ideal solution for our work. The chosen symbols are listed in the parser and the algorithm filters the symbols using a regular expression. Moreover, while executing the example code, some of the symbols appear most of the time due to contention. An algorithm to parse *perf*

Algorithm 3: Algorithm to parse JLM data into a CSV file

```
open_file;
headers ← ['GETS', 'TIER2'...];
write_file(headers);
for each timestamp do
    iterate_lines ← false;
    lines ← read_file('jlm' + timestamp);
    for each line do
        tokens ← split(line);
        stripped_line ← strip(line);
        if stripped_line == 'LEGEND' then
            iterate_lines ← false;
        end
        if stripped_line == 'Java Inflated Monitors' then
            iterate_lines ← true;
        else
            if line != ' ' and iterate_lines == true and length(tokens) > 0 then
                new_line ← join(tokens);
                ;          /* Values of GETS,TIER2 ...  etc */
                write_file(new_line);
            end
        end
    end
end
```

data is shown in the Algorithm 4.

3.3.3 Data Preprocessing and Classification

Data preprocessing and classification is the final step in the methodology. In this step, we perform some processing to our raw dataset and make our dataset ready to be classified. This third step of our methodology is divided into some more sub-steps which are described below:

Algorithm 4: Algorithm to parse *perf* data into a CSV file

```
symbols ← ['_raw_spin_lock'...];
values ← {};
counter ← 0;
for each symbol do
    values[symbol] ← []
end
for each timestamp do
    lines ← read_file('perf' + timestamp);
    lines ← find_chosen_symbols;
    temp_var ← {};
    for each line do
        name ← split(line)[2];
        sample ← split(line)[0];
        temp[name] ← {'sample_count' : sample};
        for each key in values do
            append temp[key]['sample_count'] into values[key];
        end
    end
    counter ← counter + 1;
end
open_file;
header ← join(each key in values);
write_file(header);
for x in range(counter) do
    temp_values ← [];
    for key in values do
        append values[key][x] into temp_values;
    end
    final_values ← join(temp_values);
    write_file(final_values);
end
```

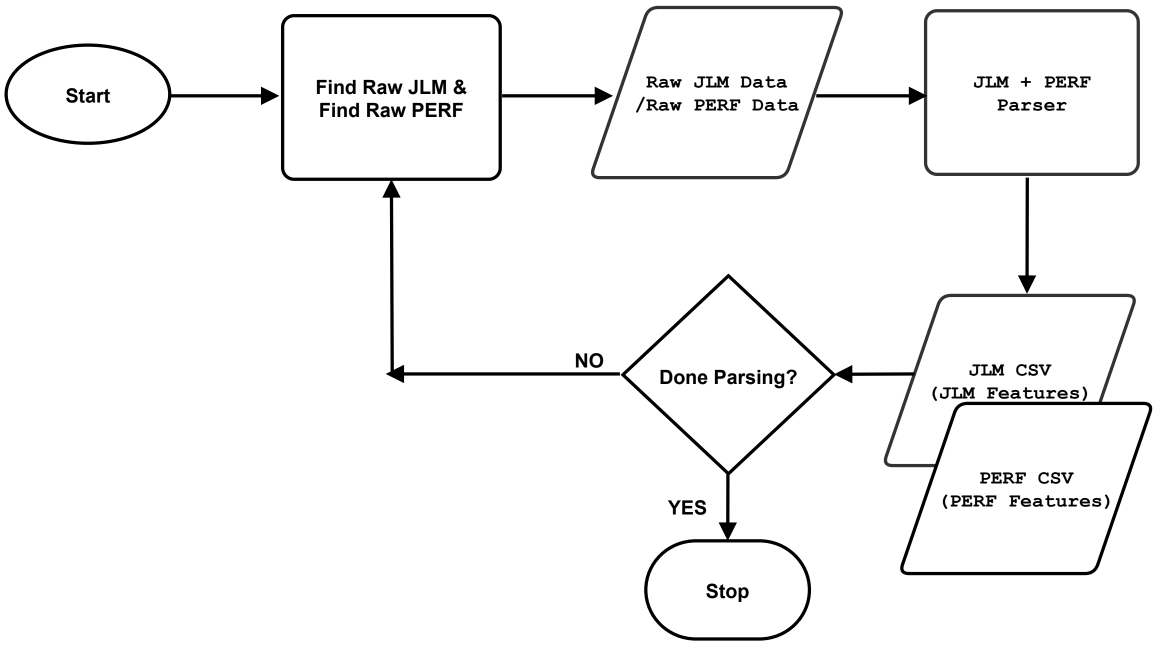


Figure 3.7: Data filtering and aggregation

Merging CSV Files

Typical to most classifiers it is important to perform some data preprocessing prior to training the classifier. Our preprocessing starts with concatenating the three CSV files into one dataset. Both the JLM and *perf* CSV data are organized based on timestamps and they are synchronized when merged. However, it is worth mentioning that, we take the help of a popular Python library **"scikit learn"** and its modules to perform all kinds of data preprocessing and classification.

Feature Engineering

Featuring engineering enhances the performance of the model and is essential for machine learning. Therefore, in this sub-step, feature engineering is performed in order to obtain a fine-tuned dataset. Initially, the data is scaled so that it is more uniformly distributed. Obtaining a better performance from an ML model is often dependent on scaling [21]. Therefore, scaling is required before feeding the data into any clustering algorithms. **StandardScaler** from scikit learn is utilized in order to perform scaling. Several popular scaler functions are available such as StandardScaler, MinMaxScaler, RobustScaler, etc. However, considering StandardScaler over MinMaxScaler does not make much difference. Both scaler functions are widely used in ML approaches. After that, leveraging the Heatmap [22] it is possible to view the relationship among the performance metrics. This heatmap technique not only provides insight into the data, but also helps reduce some of the less correlated (or they do not have any correlation at all) metrics from the dataset. Therefore, some features are filtered out based on the heatmap analysis. In order to make the information more transparent to the reader, our heatmap analysis can be illustrated as a generic example. Regarding the correlation analysis, it is visible that feature GETS, TIER2, TIER3, _raw_spin_lock (index 0, 3, 4, 7 respectively in the heatmap) are negatively correlated to feature AVER_HTM (index 6 in heatmap correlation matrix). This indicates, the lock acquisition and spin-related metrics increase when holding time decreases and vice versa. Additionally, this analysis represents that the data is cluster-able. Regarding feature reduction, heatmap points out that features GETS and NONREC (index 0 and 1 in the heatmap) are highly correlated and any of them is useful for the analysis, not both. Hence, the NONREC is removed from further analysis. The heatmap correlation matrix is shown in Figure 3.8.

Applying PCA

In the next phase of our approach, we reduce the dimensionality of the data by applying the Principal Component Analysis (PCA) [23]–[25]. This popular algorithm mainly finds the p-dimensional eigenvectors of the data's covariance matrix. As we set the required dimension equal to two, PCA helps visualize the data in the two-dimensional form as the data features are reduced to two primary components only. We extract the reduced two main components **"Principal Component 1"** and **"Principal Component 2"** after applying the PCA algorithm. In unsupervised learning, feature engineering or feature extraction is often done using this PCA analysis because it extracts the most

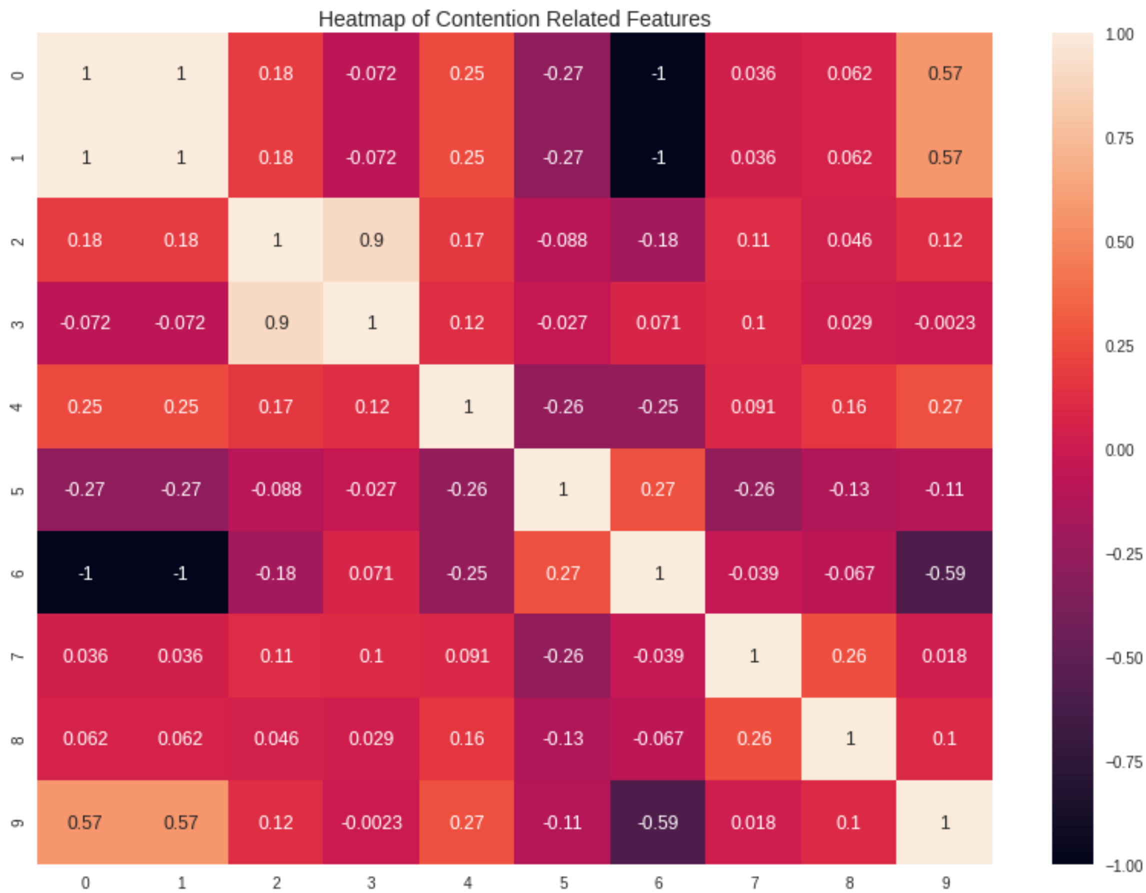


Figure 3.8: Heatmap Correlation Matrix

crucial desired N features out of a comparatively large dimension. Hence, in our case, these two primary components are the final features that are ready to be fed into the further clustering algorithms. Additionally, plotting these two dimensions in a scatter plot makes visualization more efficient and understandable.

KMeans & Cluster Analysis

Before feeding the data into a clustering algorithm such as KMeans, it is required to know the expected number of clusters. Most of the clustering algorithms need the expected number of clusters as an argument prior to the execution of the clustering process. However, this expected optimal number can be obtained leveraging some popular clustering analysis or methods. Although our desired number of clusters and our expectation from the dataset is two, we set the argument of cluster number as three for the algorithm because this optimal number of clusters is verified and extracted by the available methods, such as the Elbow method or the Silhouette Coefficients technique. After applying these techniques, they respond with the optimal number of clusters possible in our dataset. That result is the argument of cluster number for our clustering algorithm we set. Finally, the classifier can be trained using the PCA data. We feed PCA data to the KMeans with necessary arguments.

Classification process is not just limited only with the PCA values but we also feed KMeans with the processed final data to find the expected clusters of fault types. However, these two training approaches find the similar kind of results and we verify it with some performance evaluation methods discussed in Chapter 5, Section 5.7. In order to verify whether the clustering algorithms are compatible with our dataset we use another clustering algorithm named Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [26], [27]. However DBSCAN fails to construct desired clusters or groups out of the dataset. Unlike KMeans DBSCAN does not require argument for expected number of clusters. Instead it requires an argument called “eps” which is the value for dense area to compute the next neighbour. Initial value is set to 0.3 for eps and tweaking this value sometimes helps extract desired clusters. Tweaking to a higher value does not improve the clustering performance in this case of dataset. Therefore, we leave it out from our performance evaluation.

After the clustering process, we extract the labels and attach them to the original dataset to observe the dominant features for each cluster. In order to achieve this, we take the assistance of a visualization method called Radial Visualization. The Radial Visualization method is a data visualization technique to display multivariate data in a circle. This algorithm plots each feature dimension uniformly around the circumference of a circle then plots points on the interior of the circle such that the point normalizes its values on the axes from the center to each arc [28]. The radial visualization allows plotting multiple dimensions within the circle, widely exploring the dimensionality of the visualization. Data scientists use this visualization algorithm to know the classes' basic distinction or observe too many outliers. As a generic example, how a radial visualization works can be described from our radial visualization analysis taken from Chapter Clustering Results 5. The graph representation of this technique is shown in Figure 3.9. In order to visualize the data through this technique, a dataframe and the targeted column name are passed as the two primary arguments for the radial visualization method. The targeted column name is required, based on which the graph separates the classes and places them towards the dominant features around the circle. From the Figure 3.9 it can be seen that the dataset is grouped into three classes. One of the classes is gravitated towards in the middle of AVER_HTM and CTX_SWITCH, one of them towards GETS and the last one is attracted to TIER2 and TIER3. However, our detailed results of the radial visualization is discussed in Chapter 4, Section 5.6.

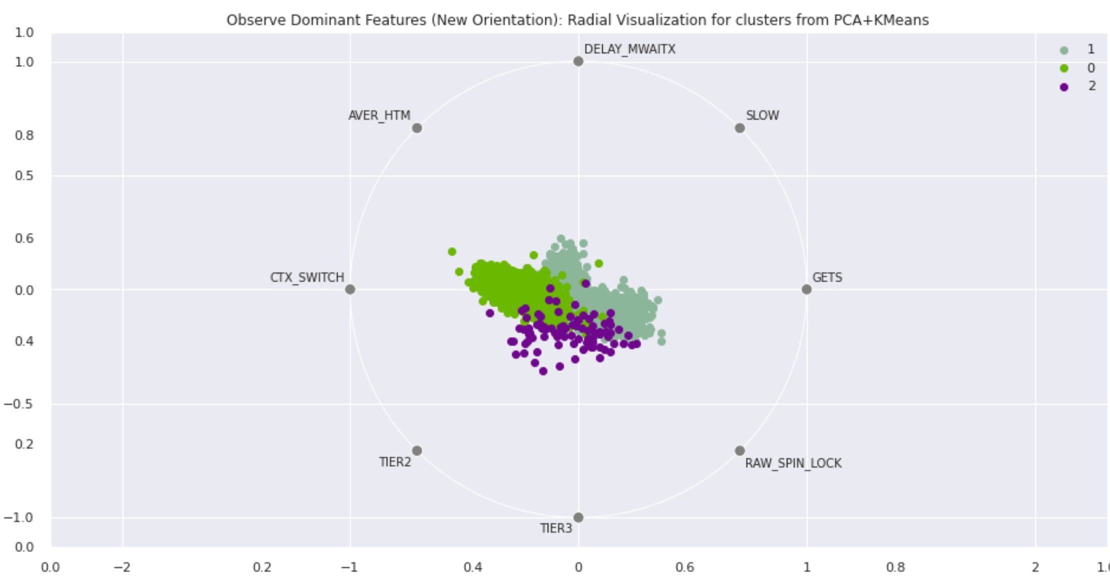


Figure 3.9: As a generic example, observing key features using 2D Radial Visualization for clusters extracted from PCA+KMeans algorithm.

Although utilizing radial visualization reveals some insight into the data, it is partially successful in distinguishing the dominant features for all the clusters. However, our analysis tries to visualize

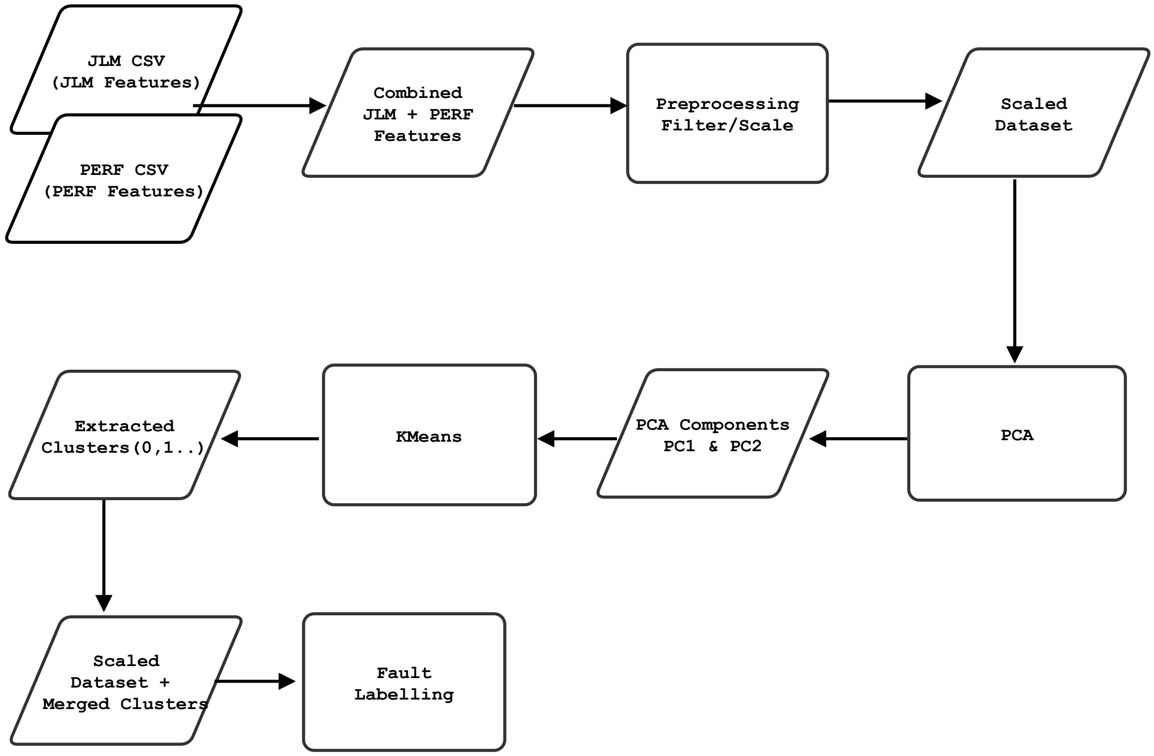


Figure 3.10: Data preprocessing and classification

the clusters changing the orientation of the features placed around the circumference of the radial visualization's circle. Organizing the new orientation does not improve the visualization performance and hence observing the dominant feature is still partially solved. Therefore, we move forward to another type of visualization method called box plot. Box plotting is a visualization method [29] that displays the data distribution based on five-point summary ("minimum", first quartile (Q1), median, third quartile (Q3), and "maximum") [30]. Plotting the features to the box plot reveals the dominant features for each cluster. However, in order to label the clusters, the test parameters (e.g., Threads and Sleep) are mapped back to the processed dataset. According to our hypothesis, one cluster should have a relationship with thread numbers, mainly fault type 2 (high-frequency access by threads). Utilizing box plots and plotting threads in relation to clusters reveals that one cluster falls under a high-frequency access fault. Plotting box plot for Sleep time also reveals that one cluster has a relation to it which is expected according to our hypothesis.

Plotting the other features with a box plot also helps us find the dominant features for each cluster. Observing different metrics utilizing box plots and the discussion of labeling fault type is presented in Chapter 6. The whole preprocessing and classification process is captured in Figure 3.10.

3.4 SUMMARY

The summary of this chapter includes a discussion about the methodology of the research work. The method is driven by a hypothesis that contention faults occurring due to a) heavy hold time inside the critical section or b) high-frequency access by threads can be classified as they leave some patterns in the metrics of the performance analyzer tools. It summarizes the chapter by discussing the main three method steps a) performance metrics acquisition, where it discusses the collection procedures of performance metrics log by exercising the concurrent code. The next step, b) data filtering and aggregation, where it discusses the procedures we take into account to filter out the required data from both the JLM and the *perf*. This step, also discusses how information is merged from multiple JLM files into one by creating a CSV and performing the same for the *perf* CSV. And finally, c) preprocessing and classification step discusses the clustering techniques we performed on our generated dataset to classify our expected clusters of fault types. It also points out the discussion of how a cluster can be labeled back to its actual fault type.

Chapter 4

DATA GENERATION

4.1 INTRODUCTION

Data generation is one of our main contributions in this research. Due to unavailability of the proper dataset it is one of our major concerns to generate dataset on which we can apply the clustering techniques. Besides generating datasets, ensuring proper hardware, machine, and software tools was another big challenge. In this chapter, we also try to mention all the tool-set, hardware, and environment needed to perform our research experiment. Moreover, the chapter describes the experimental setup, describing how we exercise our example code varying some parameters. Stress testing a concurrent Java application through a multi-core processing environment requires a high-performing machine with high memory resources. While running a concurrent application, one must keep in mind that several aspects are running underneath that often experiences overhead. Hence, it is recommended to maintain a quiet environment to run an application with concurrency. In the case of a Java application, JVM performs code optimizations. Then it operates locking mechanisms that require both space and time, and finally, in case JVM fails to manage the locks, context switching occurs between JVM and OS kernel [3]. Moreover, running java code with multiple threads need more resources than a regular java application. Thinking of all these corner cases, we try to maintain an ideal environment for our experiment. This chapter highlights the environment and experimental setup where we ran our example code. And it has the description of the tool-sets that assist in generating and processing our datasets for this research.

4.2 Environment Configuration

4.2.1 Hardware Configuration

A high-performing machine with a bare-metal operating system installed in it is ideal for generating the run-time contention performance data. Moreover, an isolated environment is also recommended for the execution of *perf* and JLM. We install these tools on a high-performing Linux machine with the following configurations:

- **CPU:**
  - Product: AMD Ryzen 9 3900X 12-Core Processor
  - Architecture: x86_64
  - CPU(s): 24
  - Frequency: 3800 MHz
- **Memory:** 32 GB

4.2.2 Java Configuration

JLM is compatible with the OpenJ9 JVM [31]. Hence, for the Java environment we use Eclipse Openj9 Virtual Machine. The Java configurations are as follows:

- **JDK:** Openjdk version “1.8.0_292”
- **JRE:** OpenJDK Runtime Environment (build 1.8.0_292-b10)

- **JVM:** Eclipse OpenJ9 VM (build openj9-0.26.0, JRE 1.8.0 Linux amd64-64-Bit Compressed References 20210421_1000 (JIT enabled, AOT enabled)

4.2.3 Performance Metrics Acquisition Tools

The following tools were used to capture the run-time performance metrics of the application:

- **Performance Inspector:** The IBM Performance Inspector is a tool-suite that includes some performance measuring tools such as *TPROF* for CPU profiling, *JPROF* for application profiling, and JLM for lock profiling. To capture contention statistics and inflated monitors information we install this performance inspector in our machine.

- ***Perf* Tool:** *Perf* tool mostly comes with the Linux distributions. In order to capture performance data and symbols from kernel space *perf* tool is installed in our machine. Couple of terminal commands are needed to install this tool in a Linux machine. We ensure proper installation of this tool during running the experiments.

- **Perf-hottest:** Data recorded using *perf* tool is stored in a file named `perf.data` by default. This data is saved in the same directory where perf record command is executed. However, the data extracted from *perf* is not human readable. The Perf-Hottest tool is used to interpret information from the "perf.data" file and translate it into a human-readable form.

4.2.4 Log Generation : Manual Steps

Data generation can be done in a manual way where human intervention is more active to operate the procedures until collecting the data. It requires an operator to be more attentive and needs careful frequent terminal switching to capture the exact data. Operating the manual data generation and collecting the data, we need the performance analyzer tools to be appropriately installed prior to the code exercise. However, in this manual way, we start the java application in one terminal, providing necessary profiling arguments (e.g., "-Xjit:perfTool", "-agentlib:jprof") for the concurrent java application, and in another terminal, we record the *perf* data using the java pid. Extracted raw "perf.data" is converted to a human-readable file leveraging the "perf-hottest" python script. Later, we open another terminal and start the JLM agent to capture contention-related statistics data, which is a raw JLM file. Before running the java application and capturing the other data, we ensure the proper installation of those tools that are described in the above sections. Installation of these tools is described below step by step.

- **Install Adoptopenjdk:** Installing Adoptopenjdk ensures installation of OpenJ9 JVM which we need for our data generation procedure. From the adoptopenjdk download page we download the compressed JDK file and extract it to the desired path of our Linux machine where most of the other versions of JDKs are installed by default. We add this java home path to the `$JAVA_HOME` environment variable. The following terminal commands ensures adoptopenjdk installation.
  $ sudo mkdir -p /path_to_java_home/
  $ sudo cp /home/$USER/Downloads/jdk-11.0.8+10.tar.gz /path_to_java_home/
  $ cd /path_to_java_home/
  $ sudo tar -xvzf jdk-11.0.8+10.tar.gz

- **Install Perf Tool:** *Perf* tool comes with the Linux distribution most of the time but we ensure it is installed in case of unavailability. In order to install perf tool following commands are used:
  $ sudo apt install linux-tools-common
  $ sudo apt install linux-tools-generic
  Although, perf tool captures kernel memory trace, which is not permitted for the first time from any other user except root after enabling the perf tool. It is required to grant the permission by changing the kernel settings. This permission is enabled by the following commands:
  $ sudo sysctl -w kernel.perf_event_paranoid=1

$ sudo sysctl -w kernel.kptr_restrict=0

- **Install Performance Inspector:** IBM performance Inspector provides necessary tools such as JPROF, TPROF, or JLM to profile java applications and check their health. In order to capture contention-related data, we ensure installing this tool-suite in our machine. As this tool-suite is a proprietary product and managed by IBM internally, we finally get access to the product with the help of the IBM team. However, after unpacking the product, we need to build this tool-suite to enable all the other modules of it. Building it requires modern C++ compilers and Linux headers and libraries. The prerequisites to install performance inspector are:

  - cmake
  - binutils-dev
  - libiberty-dev

  These are installed before installing the tool-suite. Once those are done we place the inspector package to a particular directory of the system. Then we create a new directory called `build` inside the root directory of the suite. Next we perform `cmake` and `make` command to build the inspector. This installation adjusts a `bin` and a `lib` directory under the inspector root. The next requirement for this tool is to attach the `bin` directory to the environment `$PATH` variable and the `lib` directory to the library path of the system. The following commands do all the processing for us:

  $ export PATH=/path_to_ibm_pi/bin:$PATH

  $ echo "/path_to_ibm_pi/lib" | sudo tee /etc/ld.so.conf.d/ibm.pi.conf

  $ sudo ldconfig

4.2.5 Log Generation : Automated Steps

Generating the data and the complete dataset is a time-consuming process, and reducing this amount of time turns us to automate the process. It takes around 40 to 45 seconds to perform a single run and generate the JLM & *perf* data and store them. Moreover, it needs more human involvement when the data generation process is operated manually. Therefore, we take the help of more than one bash script algorithms capable of exercising the code and the rest of the other processing until saving the data in a desired location of the machine. The algorithms are listed in Algorithm 1 and 2. To assist us in generating the dataset with ease, we wrote an algorithm (steps listed below) capable of running the entire process multiple times in our configured environment.

- Set thread number and sleep time
- Run java program
- Wait ten seconds
- Execute *perf* and JLM for ten seconds
- Terminate java program
- Collect PERF and JLM data
- Repeat n times

4.3 Dataset Creation

Starting from clustering till training a classifier, contention performance data is required to classify the contention fault types. Unfortunately, this performance data is not readily available, and dataset generation was another concern we had to attend to. However, concurrent codes that create the worst contention are not many, like those that make different bugs. Instead, code with an extensive critical section or high access frequency to the locked resources can cause contention issues, and

Listing 6 Java Synchronized Task example emulates types of faults

```java
class SyncTask {
  public Set<String> set;
  public int sleep_t,

  public synchronized void taskOne(String value){
    try {
      set.add(value);
      Thread.sleep(sleep_t);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void taskOneV2(String value){
    synchronized(set){
      try {
        set.add(value);
        Thread.sleep(sleep_t);
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

a simple synchronized block may produce bottlenecks. Hence, in our experience, we encountered many example codes related to synchronization but ended up observing similar patterns and are less extendable. Any faulty concurrent code pattern can be mapped into a critical section with the high computational operation or accessed by the threads with high frequencies. In order to emulate some levels of contentions, we consider an example of concurrent code. The class that emulates the contention faults is shown in the Listing of 6. The driver class that initiates and controls the execution of the thread is shown in the Listing 7. For simplicity, our examples implemented the synchronized instance method only.

Due to the absence of the dataset, we had to focus on the dataset generation process as well.

Listing 7 Java Synchronized Task driver class example controls the thread execution

```
public class SyncTaskMain {
  public static void main(String[] args) {
    int NUM_THREADS = thread_size;
    Set<String> set = new HashSet<String>();
    SyncTask sl = new SyncTask(set);
    ArrayList<Thread> threadList = new ArrayList<Thread>();

    for(int i = 0; i < NUM_THREADS; i++){
      Thread t = new SyncTaskThread(sl);
      threadList.add(t);
      t.start();
    }

  }
}
```

Although the major portion of our research is spent on dataset generation, we successfully overcame this situation in the end. We intended to move forward in an unsupervised way, and because of this, our generated data was unlabeled. We applied different clustering algorithms and mapped the classified data to the original one to strengthen our assumption.

However, we divide and formalize the test scenarios for our dataset generation process and execute concurrent code with various test parameters configurations such as multiple threads and different sleep times along with slightly modified code to emulate the contention scenario.

4.3.1 Test Formalization for Dataset

In our work we try to formalize the test scenarios in such a way that it can experience some levels of contention as well as two different types of contention faults we focus in our work **a) high hold time, b) high frequency requests by the threads**. We varied in the exemplar code, the time spent in the contended region (Sleep time), as well as the number of threads. Our formalized scenarios for the data generation process is given below:

We configure a bash script algorithm that takes or generates the necessary values for running the java example code. The first loop of the bash script algorithm iterates through an array of thread values. We operate the whole data generation process in multiple configurations and each configuration we take a set of threads in the array. The threads are listed in each phase below. Inside the first loop, our second loop iterates through two hundred different sleep times. Our multiple values are being used as the run-time arguments for the java example code and that emulates the different levels of contentions including contention with high hold times and high frequency access to the locked resources by threads.

- **Configuration 1:**
  - Threads: [10, 100, 500, 1000]
  - Sleep: From $1ns$ up to $20,000ns$ in $100ns$ increments, with 200 runs
  - Total Data Points: 800
- **Configuration 2:**
  - Threads: [10, 100, 200, 300, 400, 500, 1000]
  - Sleep: From $10ns$ up to $20,000ns$ in $100ns$ increments, with 200 runs
  - Total Data Points: 1600

There is a gap between the two different sleep times we configured, and that is 100 nanoseconds. Leaving this much gap between the two different sleep times is intentional, and it allows us to cover a wide range of sleep times. Collecting data after running a single test is also a time-consuming task, and it takes 40 - 45 seconds to complete one data point collection.

In order to emulate the numerous levels of contention such as high contention, low contention, or high-frequency requests by threads, we use a various range of sleep times inside the critical section and low & high range of thread numbers, respectively. However, the large amount of threads mainly maintains a routine to send access requests to locked resources at the same time that ensures high-frequency access requests to the critical section. In order to perform perf recording and JLM recording, our example code needs to be running for more than twenty seconds. Therefore, from the inside of the run method of the "SyncTaskThread" class, we maintain a tight loop that ensures running the program a little longer. In this case, the value for this loop is set to 100000. Regarding the sleep time configuration, we only keep the time in the nanoseconds range. Two different code execution configurations are considered to generate the whole dataset, which aggregates a total of twenty-four hundred data points. However, the second configuration contains some extra thread numbers in its thread array. Other than that, the two configurations are executed with similar parameters. The intention behind running these two configurations is to increase the data points for our dataset.

Features	**Analyzer Tool**	**Data Type**	**Features**	**Analyzer Tool**	**Data Type**
%MISS	JLM	Numerical	REC	JLM	Numerical
GETS	JLM	Numerical	%UTIL	JLM	Numerical
NONREC	JLM	Numerical	AVER_HTM	JLM	Numerical
SLOW	JLM	Numerical	_raw_spin_lock	PERF	Numerical
TIER2	JLM	Numerical	ctx_sched_in	PERF	Numerical
TIER3	JLM	Numerical	delay_mwaitx	PERF	Numerical

Table 4.1: Lock-Contention Performance Metrics dataset information.

4.3.2 Dataset Information

The final generated dataset comprises **twenty-four hundred** data points, and **twelve** features in total. Features %MISS, GETS, NONREC, SLOW, TIER2, TIER3, REC, %UTIL and AVER_HTM are collected from JLM. Features "_raw_spin_lock", "ctx_sched_in" and "delay_mwaitx" are collected from *perf* tool. All of them are numerical data. The dataset's features, analyzer tool it comes from and the data type are shown in Table 4.1.

4.4 SUMMARY

This chapter summarizes the data generation procedures through some sections that discuss the environment configuration and tools installation required for data generation for this specific research. A quiet and powerful machine is recommended to perform the operation that helps in generating the dataset. We ensure a machine with high configuration, and it has necessary java and kernel performance profiling tools installed such as JLM, *perf*, IBM performance inspector, OpenJ9 JVM, and JDK, and some Linux common tools. Moreover, we ensure that no other processes are running except the bare-metal Linux OS. Different processes may hamper the concurrent java code's execution time, reflecting the performance metrics. This chapter also summarizes both manual and automated steps to generate the dataset. It also includes the information of test formalization and configuration that helps generate a dataset of contention statistics-related metrics. We end this chapter by summarizing the information of the dataset, such as the total number of data points and features and the features types.

Chapter 5

CLUSTERING RESULTS

5.1 INTRODUCTION

The expectation is to apply several clustering techniques and later observe different clusters, including the two potential fault types within the dataset. However, before utilizing the clustering techniques, a top-level data analysis is accomplished during the execution of our example code. This top-level investigation helps us understand metrics changes based on fault types and internal connections among the performance metrics. Later, we move forward with the process of an advanced data analysis for the clustering, which is the main focus of this chapter. The importance of advanced data analysis is to help us find internal connections among the features and cluster-able quality of the data. Also, this analysis assists us in accelerating further analysis by reducing some unnecessary features from the final dataset. In machine learning, it is essential to extract impactful features because the unnecessary features increase the chance of performing a model under-fit, or often over-fit [32]. We elaborate preliminary data analysis by observing both the JLM and perf data and discussing the internal connections between some features. Later, we analyze the correlation matrix and reduce the features by plotting a heatmap. Additionally, the initial analysis of the data yields information regarding clustering tendency, which means how good our data is to be clustered. Moreover, it assists us in finding the expected optimal number of clusters which is a required parameter for the clustering algorithms.

Throughout the chapter, the initial data analysis and the results of the clustering processes and the obtained clusters are discussed.

5.2 An Initial Observation of Metrics & Correlations

A plain eye observation on the JLM data is often helpful in understanding the metrics changes based on different contention cases. With regards to non-contention or less contention, the JLM metrics never appear or often appear with low lock competition degree (spin counts) and low average monitor hold time. Our investigation finds that monitor entries appear on the JLM data with a high spin count (e.g., GETS, SLOW, TIER2, TIER3) when there are an increased number of requests to the locked resource by the threads. On the other hand, the metrics come with a high average hold time (e.g., AVER_HTM) when the threads hold the lock for more than expected.

In order to move forward with unsupervised learning and prove our hypothesis, it is required to find some correlations among the data points. Machine learning is, after all, data-driven AI, and our model will be as good or as bad as the data we have [32]. Although some studies listed under Chapter 2 state that there are no correlations among or between the features of JLM, our careful observation finds out some interesting insights. JLM metrics do change based on the fault types, and once the metrics related to a particular fault are affected by that fault type, some impacts are observed on the other metrics at the same time. In our emulation, while executing our example code, we run some scenarios where an operation holds a lock for an excess amount of time. In this scenario, we observe that the AVG_HTM of a monitor increases while the lock acquisition or spin-related metrics (e.g., GETS, TIER2, TIER3) decrease in number. In contrast, the metrics related to hold-time decrease while the metrics related to spin count increase in number. However, our preliminary top-level investigation finds that a lock usually experiences access requests by the threads with the highest frequency only when they spend a shorter period inside the critical section. It implies that, at the same time, when a lock experiences high hold time and high-frequency access requests, the high hold time conquers the overall situation. Therefore, to be concluded, the hold time feature is a dominating feature over the high-frequency access requests.

Based on these observation, the following statements can be constructed:

1. If threads hold the lock for more than expected, metrics related to hold-time (e.g., AVER_HTM) increase in number, and metrics related to spin count (e.g., TIER2, TIER3, _raw_spin_lock) or lock acquisition (e.g., GETS) decrease in number and vice versa.

2. When the two faults (e.g., Fault-1 and Fault-2) occur at the same time, the metrics related to spin count increase in number only when the threads spend shorter period of time inside the critical section.

5.3 Data Preprocessing

A correlation heatmap is a data plotting that helps visualize the data and expresses the inner connections among the features. Leveraging the heatmap, data scientists often find insight into the data, such as features that are positively correlated or negatively correlated to each other. Therefore using the data to generate a heatmap, we can observe the internal connections between each of the features. The heatmap correlation for “SyncTask” example is shown in Figure 5.1. The heatmap plotting is done using our final dataset after merging the *perf* and JLM data. Also, the features that we use in the heatmap are listed in an index table (see Table 5.1), as the heatmap is organized using numerical indices. A heatmap analysis of the performance metrics (see Figure 5.1), finds an interesting correlation among the metrics related to lock acquisition, such as GETS, metrics related to spin counts such as TIER2, TIER3, _raw_spin_lock and lastly, metrics related to hold-time such as AVER_HTM columns of the data. The investigation observes that metrics related to spin count and lock acquisition have a positive correlation among them. On the other hand, a negative correlation is observed between the hold-time metric and those related to spin counts. This observation strengthens our belief that the performance metrics could be classified. Also, analyzing this heatmap plotting assists us in sorting out some important features that remain within the dataset until the final stage of the clustering processes.

Index	Label
0	GETS
1	NONREC
2	SLOW
3	TIER2
4	TIER3
5	%UTIL
6	AVER_HTM
7	_raw_spin_lock (RAW_SPIN_LOCK)
8	ctx_sched_in (CTX_SWITCH)
9	delay_mwaitx (DELAY_MWAITX)

Table 5.1: Lock-Contention Performance Metrics Indexes

Before running clustering algorithms, some data preprocessing is required. For this, we write an algorithm that parses the run-time data into the form we need, and it creates the CSV file using the values from the raw *perf* and JLM data. After filtering out the required data into CSV, analyzing the heatmap correlation matrix helps us to understand insight into the data. It illustrates that some features are related; for example, the columns GETS and NONREC is highly correlated, so we keep GETS in our dataset. The column %MISS is all zero values. Hence, we leave out this column. In order to increase the readability of the column names, some of them are renamed that are mainly collected from perf data, such as "_raw_spin_lock", "ctx_sched_in" and "delay_mwaitx" are renamed to "RAW_SPIN_LOCK", "CTX_SWITCH" and "DELAY_MWAITX" respectively. The KMeans algorithm requires data to be numerical and tabular. We perform the following steps in our data preprocessing stage to achieve this.

1. Remove unnecessary alphabetical columns.
2. Merge the *perf* and JLM into one data-frame.
3. Remove columns that have 0 values.

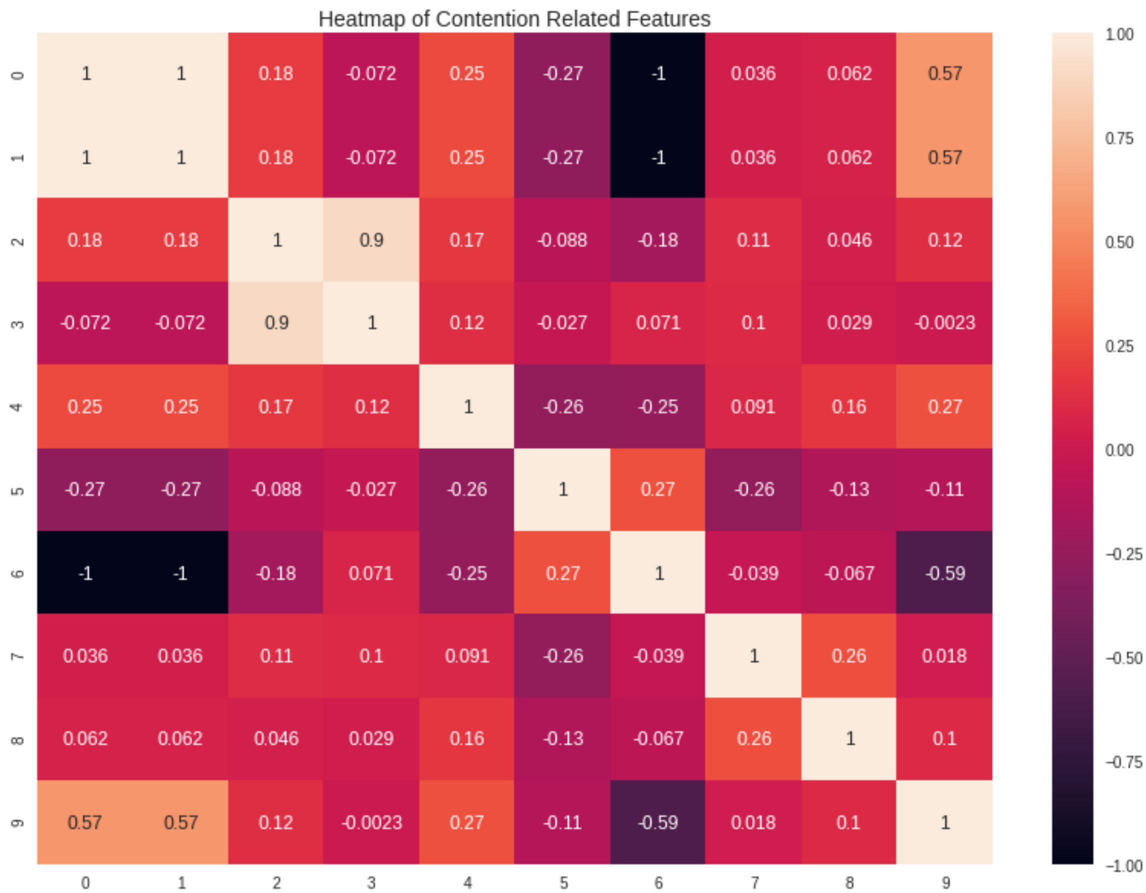


Figure 5.1: Metrics Correlation Heat-map

4. Remove unnecessary columns from the dataset after analysing the heatmap.

5.4 Optimal Number of Clusters

The most crucial part of unsupervised machine learning is analyzing the data and validate clustering tendency prior to the clustering and validating the clustering results after that. It is required to validate the clustering tendency to confirm that the data is well cluster-able. Most of the clustering algorithms normally return with some clusters even if the data does not contain any clusters or groups [33]. Therefore, two factors are important in validating clustering approaches **a)** *Assess the clustering tendency before the analysis* **b)** *validate the quality of the clustering results*. In this section we try to validate the clustering tendency and possible optimal number of clusters with the following three techniques:

1. **Assess Clustering Tendency:** This technique determines whether the dataset contains meaningful clusters.

2. **Relative Clustering Validation**: This technique evaluates the structure of the clustering process by varying different parameters of the same clustering algorithm. This technique is useful to determine optimal number of clusters can be found in the dataset

3. **External Clustering Validation:** This technique compares the results with the externally known results. This validation assists in determining the appropriate clustering algorithm for the chosen dataset.

5.4.1 Prepare Environment

Validation measurement of the clustering is done leveraging the popular R programming language, and its packages such as **cluster**, **factoextra**, **NbClust**. Therefore we prepare the environment and ensure the R language is installed in our machine as well as the r-studio, a popular IDE for R developers. When the installation is done, we perform the necessary processing to analyze the results with the above three validation techniques. However, before starting the analysis, our raw dataset is processed first with the Python data-frame, scaled the entire dataset, and exported to a CSV file. We start our analysis in r-studio, loading the CSV data utilizing the R packages.

5.4.2 Assess Clustering Tendency

Typically after applying clustering algorithms on a given dataset, all the clustering algorithms return with clusters even if the data does not contain any meaningful clusters [33]. Therefore, it is mandatory for us to determine whether the data can be partitioned in meaningful groups. In order to achieve that there are some popular methods available such as **a) Hopkins Statistic (Statistical**

Method) and **b) Visual Assessment of Cluster Tendency (Visual Method)**. However, applying Hopkins Statistics results in 0.90 for our dataset which is more than 0.5. A good clustering tendency requires Hopkins Statistic value more than 0.5. Therefore, based on this Hopkins Statistical method's result our dataset it highly cluster-able. Hopkins Stat result is shown in Figure 5.2

```
> df = read.csv("./trace4.3.5/trace4.3.5.csv")
> head(df)
  X        GETS      NONREC      TIER2      TIER3        SLOW     X.UTIL    AVER_HTM RAW_SPIN_
1 0  0.830803679  0.830803679 -1.4602945 -0.9336935 -0.05972464 -2.1998673 -0.87606794     2.240
2 1  1.399911505  1.399911505  1.9418446  1.7547678  0.69646072  0.4545729 -1.33363078    -0.522
3 2  0.362159661  0.362159661 -0.8245812 -0.5275893 -0.05972464  0.4545729 -0.44117837     0.081
4 3  0.008151021  0.008151021 -0.6973483 -0.1308946  0.94852251  0.4545729 -0.09484200    -0.522
5 4 -0.182603932 -0.182603932 -0.1699846 -0.3111693  0.44439894  0.4545729  0.09651487     0.081
6 5  1.558464721  1.558464721 -0.5304162 -1.6614937 -0.31178643 -2.1998673 -1.46448009     0.335
> # Compute the number of clusters
> keeps <- c("GETS", "TIER2", "TIER3", "SLOW", "AVER_HTM", "RAW_SPIN_LOCK", "CTX_SWITCH", "DELAY_
> df <- df[keeps]
> df_scaled <- df
> # Compute Hopkins statistic for lock-contention data-set
> res <- get_clust_tendency(df_scaled, n = nrow(df_scaled)-1, graph = FALSE)
> res$hopkins_stat
[1] 0.9040416
```

Figure 5.2: Hopkins Statistic's result shows clustering tendency for our dataset and it is highly cluster-able.

5.4.3 Relative Clustering Validation

In unsupervised machine learning it is required to obtain the clusters in the dataset and it is also required to obtain the optimal number of clusters prior to obtaining the clusters. In this relative clustering validation technique, determining the optimal number of clusters is the primary step that can be done using some popular methods such as **a) Elbow Method** [34], **b) Silhouette Method** [35] and **c) Gap Statistics Method** [33].

Elbow Method (Python & R validation):

To identify the actual optimal number of clusters in our dataset, we plotted the relationship between the number of clusters and within Cluster Sum of Squares (WCSS), which determines the number

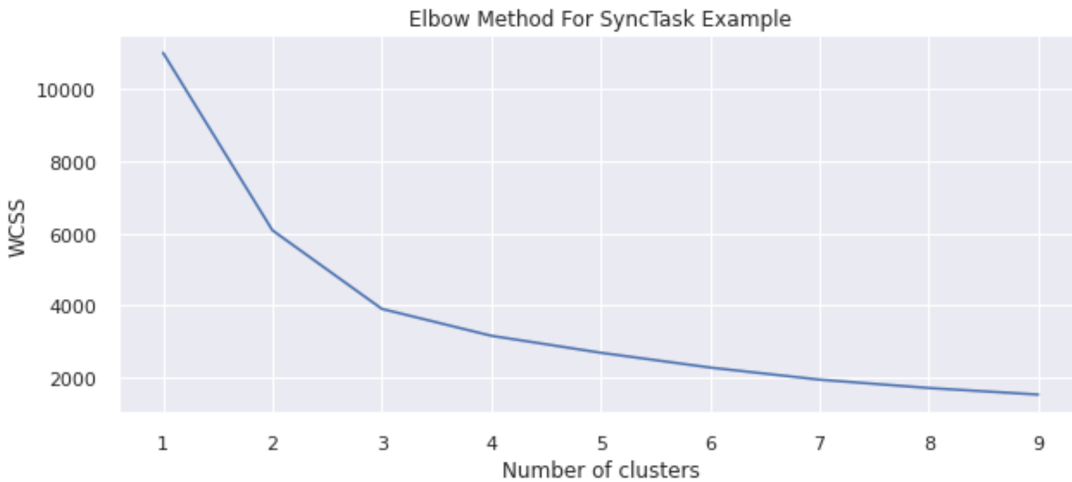


Figure 5.3: Applying K-Means and Elbow Method to obtain possible optimal number of clusters.

of the actual clusters [34]. Optimal number is determined where the change in WCSS begins to level off. WCSS is defined as the sum of the squared distance between each cluster member and its centroid. WCSS is calculated varying the k (expected cluster number) parameter of the KMeans algorithm and storing the model's `inertia_`.

After plotting the WCSS and observing it, a sharp bend at cluster 2 and 3 is visible. Either of this two number is the expected optimal cluster number for our dataset. Although the sharp bend is visible, it is often difficult to visualize the sharp bend and the elbow point, which needs a programmable calculation. We verified choosing the elbow point of the curve leveraging a Python package, `kneed` [36] [37]. The function `KneeLocator` from the package `kneed` finds out the optimal cluster number in our case is 3. The Elbow method plotting showing the optimal possible cluster number is shown in Figure 5.3. In this figure, it is visible that, y axis plots WCSS score and x axis represents number of clusters (k).

Extracted optimal number of clusters leveraging the R package is 4 for Elbow Method which is shown in Figure 5.4.

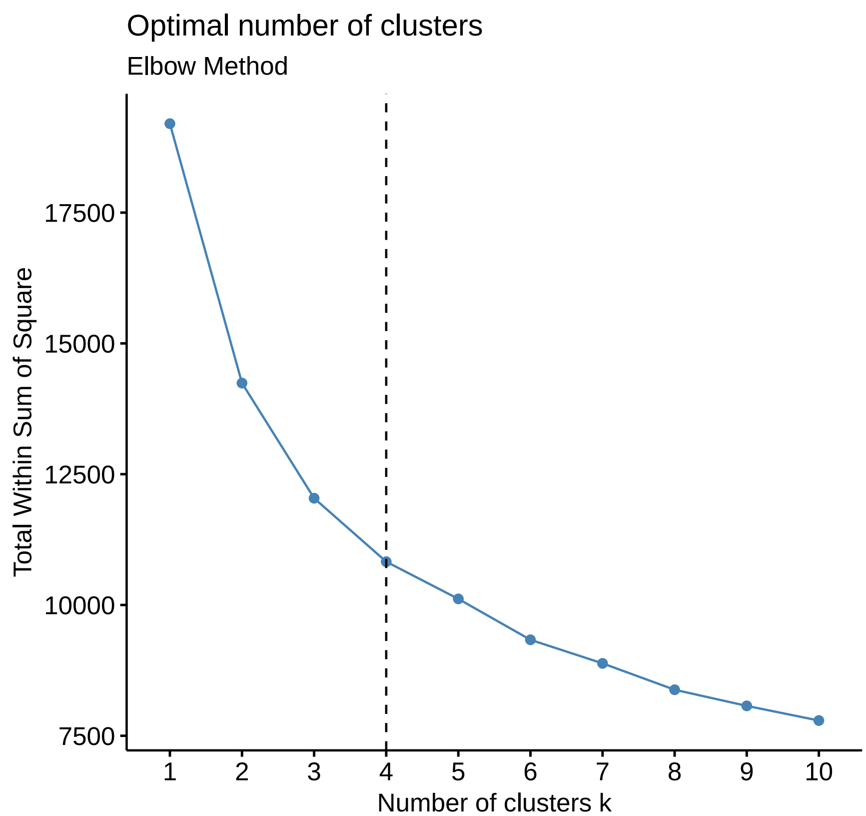


Figure 5.4: R plot of Elbow Method determines optimal number of clusters, which is 4.

Silhouette Method (Python & R validation):

A more advanced algorithm compared to Elbow method to determine the optimal number of clusters in a given dataset is Silhouette Method [35] [33]. The silhouette coefficient is a measurement of cluster cohesion and separation. This method helps decide the assignment of the data points to their proper cluster and how well the data point fits into the assigned cluster. Based on the following two factors, this assignment is done [38]:

1. How close the data point is to other points in the cluster

2. How far away the data point is from points in other clusters

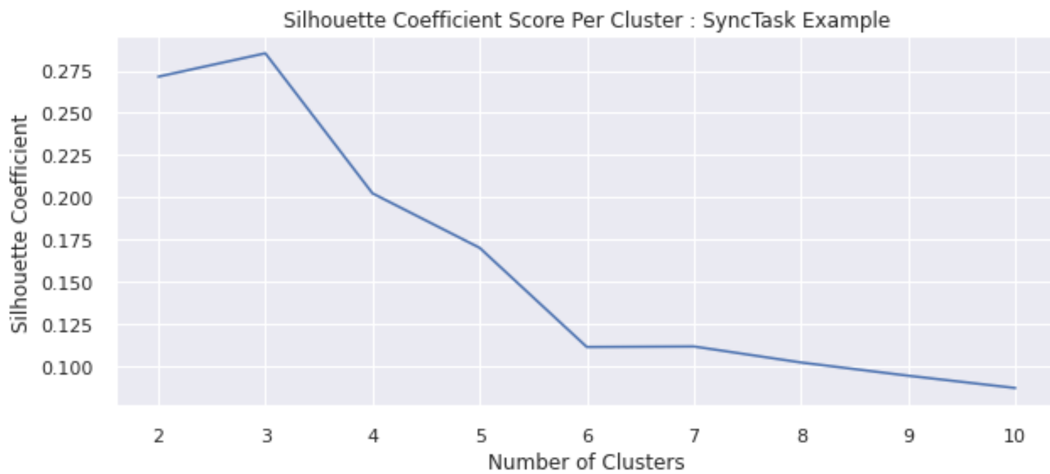


Figure 5.5: Applying K-Means and Silhouette Coefficient to obtain possible optimal number of clusters

We perform python implementation of Silhouette Method to verify the possible optimal number of clusters. Silhouette coefficient values range between -1 and 1. Higher numbers indicate that samples are closer to their clusters than they are to other clusters. The Python function `silhouette_score` from sci-kit learn `sklearn.metrics` helps us to apply silhouette scoring. However, the implementation of sci-kit learn-based silhouette coefficient summarizes the average silhouette coefficient from all samples into one score. The scoring function takes a minimum of two clusters as an argument; otherwise raises an error. We maintain the proper arguments while calculating the silhouette coefficient.

Similar to Elbow Method, we train multiple KMeans models varying the parameter **K = (expected number of clusters)** and compute the Silhouette's score for each of them. Figure 5.5 shows the optimal number of clusters which is 3 for our dataset. The score for the cluster number 3 which is the highest. The R implementation of Silhouette Method also determines that the optimal number of clusters for our dataset is 3. R plot of Silhouette Method to obtain optimal number of clusters is shown in Figure 5.6. In both Python and R implementation of Silhouette Method show that y axis plots Silhouette score and x axis plots number of clusters k.

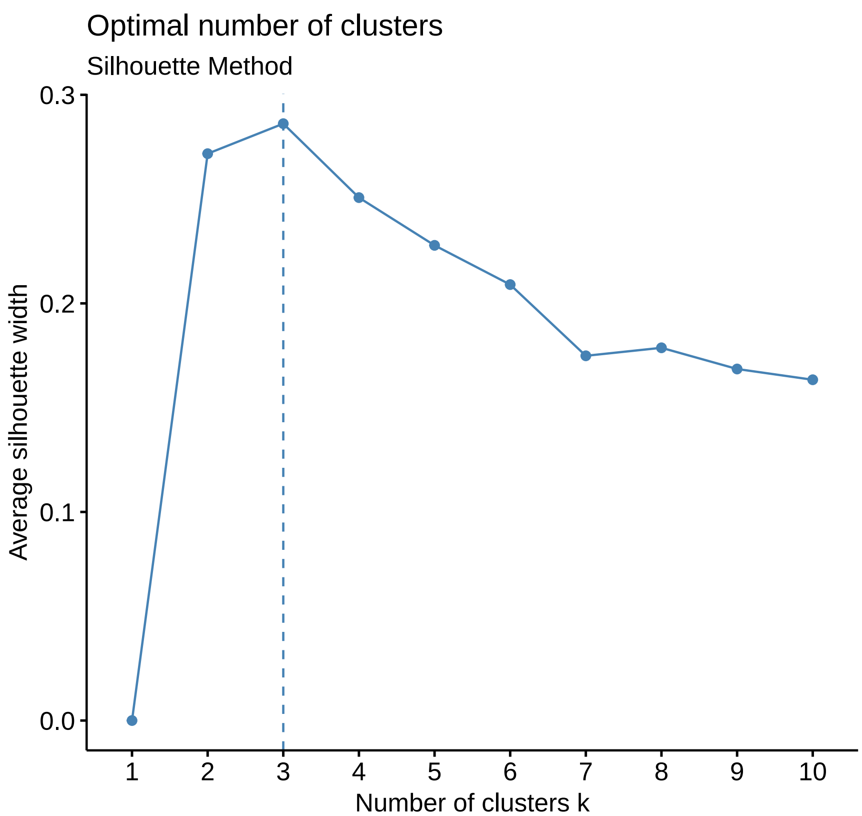


Figure 5.6: R plot of Silhouette Method to determine optimal clusters number; which is in this algorithm is 3.

Other Methods (R validation):

"Gap Statistic" is another popular method used to find the optimal value for k and has been used for more than twenty years. This method can be used for any clustering algorithm and finds the total within intra-cluster variation (W_k) for each expected cluster number. The largest W_k for a cluster number is the expected optimal number of clusters possible within the dataset. The extracted optimal number determined by the "Gap Statistic" method is 2. The R plot of Gap Statistic Method's result is shown in Figure 5.7. The y axis for the gap statistic method plots W_k, and the x axis is always k = expected number of clusters.

In order to determine optimal number of clusters, more than thirty indices has been published

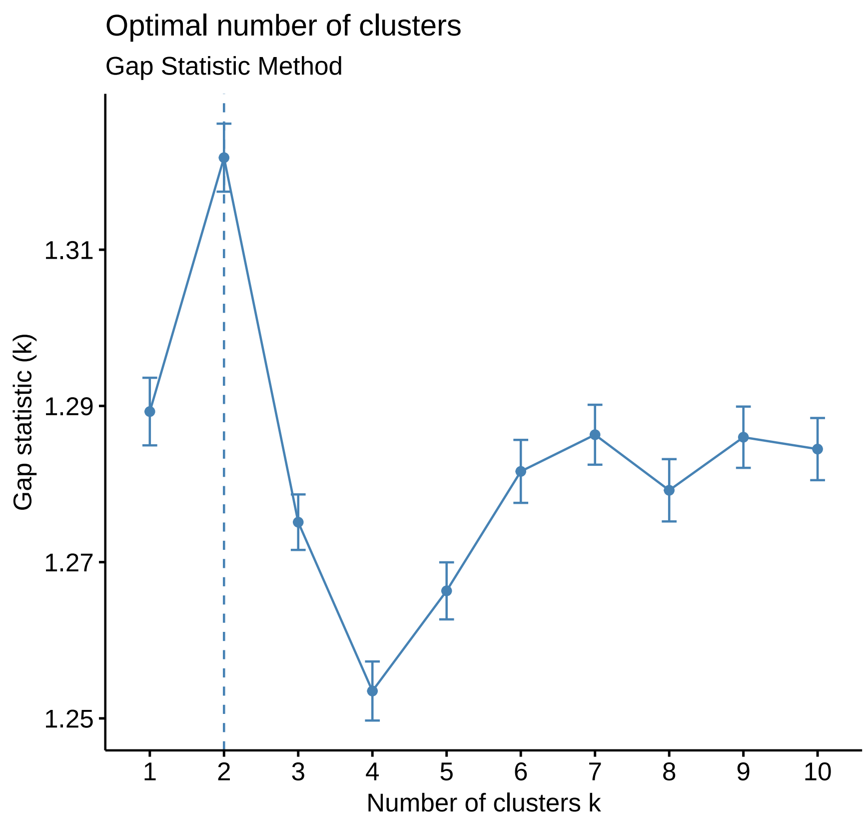


Figure 5.7: R plot of Gap Statistic Method to determine optimal cluster number; which is in this algorithm is 2.

in the literature and the R package `NbClust` [39] has aggregated them in one function. Leveraging this package it is also possible to determine the right number of clusters as the function calls all the thirty indices or methods to obtain the right number of clusters. Among all the indices 11 suggested that the number is 3 for our dataset. The r-studio console result and one of the indices called Hubert-index and last of all the plot of suggestions are shown in Figure 5.8, Figure 5.9 and Figure 5.10 respectively.

Analyzing all the methods, we come to a complete conclusion that the optimal number of clusters in our dataset is 3 and the argument k = expected number of clusters for any clustering algorithms can be set to 3 after obtaining this result.

```
> nb <- NbClust(df_scaled, distance = "euclidean", min.nc = 2,
+               max.nc = 10, method = "kmeans")
*** : The Hubert index is a graphical method of determining the number of clusters.
                In the plot of Hubert index, we seek a significant knee that corresponds to a
                significant increase of the value of the measure i.e the significant peak in Hubert
                index second differences plot.

*** : The D index is a graphical method of determining the number of clusters.
                In the plot of D index, we seek a significant knee (the significant peak in Dindex
                second differences plot) that corresponds to a significant increase of the value of
                the measure.

*******************************************************************
* Among all indices:
* 5 proposed 2 as the best number of clusters
* 11 proposed 3 as the best number of clusters
* 1 proposed 4 as the best number of clusters
* 2 proposed 6 as the best number of clusters
* 1 proposed 9 as the best number of clusters
* 3 proposed 10 as the best number of clusters

                   ***** Conclusion *****

* According to the majority rule, the best number of clusters is  3
```

Figure 5.8: R Studio console shows the statistics of determining the optimal number of clusters among all 30 indices.

5.4.4 External Clustering Validation

Choosing the appropriate clustering algorithm for a given dataset is as important as finding the correct number of clusters in unsupervised machine learning. The means of external clustering validation is selecting the appropriate clustering algorithm which fits the best for a given dataset. In order to achieve the results, one should measure the clustering statistics of different algorithms to the known results, which are the true labels of the classes. The labels for our dataset are absent, and the labeling is not possible prior to the classification. Hence this external clustering validation is not applicable in our work. Instead of analyzing the different algorithms, we choose KMeans and apply enhanced clustering known as **eclust** from R package **factoextra** helps us partition the data into three different classes. The result and plotting of enhanced clustering (eclust) is shown in Figure 5.11.

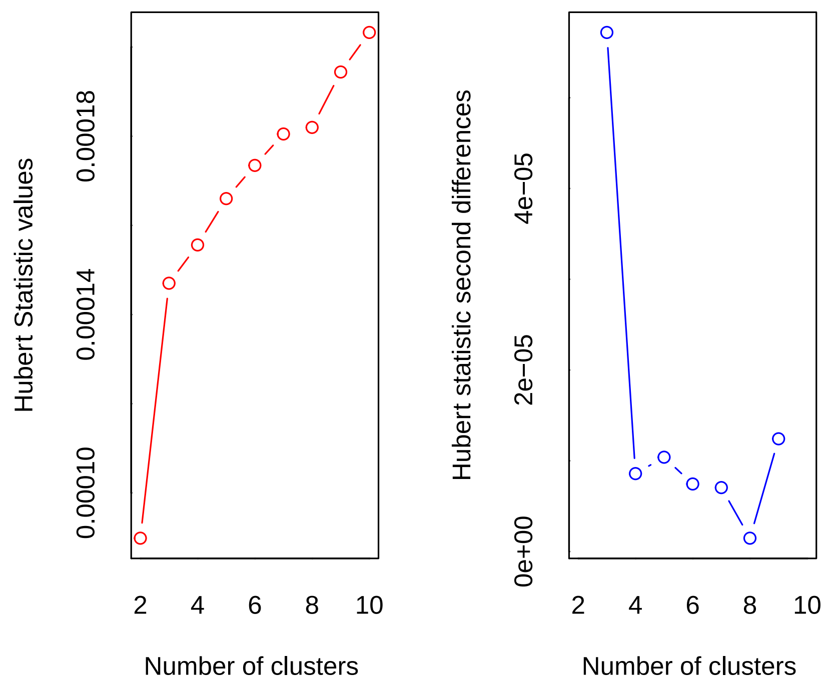


Figure 5.9: R plot of hubert-index, one of the thirty indices shows right number of clusters possible in our dataset.

5.5 Running Clustering Algorithms

Our clustering process comprises applying several clustering algorithms such as PCA (Principal Component Analysis), KMeans, DBSCAN. These clustering techniques help us to find the hidden clusters within the dataset. Using the Principal Component Analysis (PCA) method, we reduce the dimensions of our data. To achieve that, we use python library `PCA` from `sklearn.decomposition`. As we define the final output components as two, PCA outputs the two principal components out of ten starting attributes. To obtain a better result, PCA recommends scaled data, and we ensure that also using the python library `StandardScaler` from `sklearn.preprocessing`. However, PCA extracts an array of (2400,2) shaped data, which is

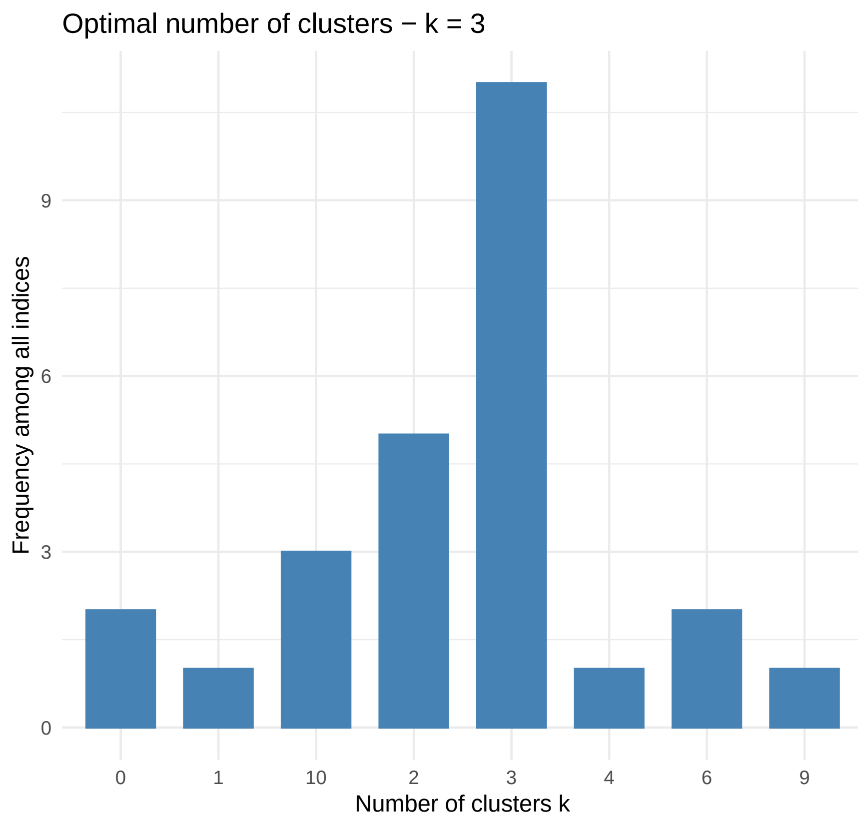


Figure 5.10: R plot of suggestions of thirty indices to obtain optimal number of clusters possible in our dataset; 11 suggested the number is 3.

appropriate to be fed into the KMeans algorithm. At this stage, KMeans locates the cluster centroids and those can be seen from Figure 5.12. As we expect three clusters based on the optimal number of clusters, then running the KMeans with the required argument, the demanded cluster number, we find the desired clusters out of the whole dataset. The applied PCA and the KMeans cluster centroids and clusters plotting are shown in Figure 5.12 and Figure 5.13. The Python library `KMeans` from `sklearn.cluster` helps us run the clustering KMeans method after the PCA dimension reduction approach.

Three red dots that are the cluster centroids can be observed from Figure 5.12. Although the down two centroids have a dense population, the upper one has a comparatively fewer population

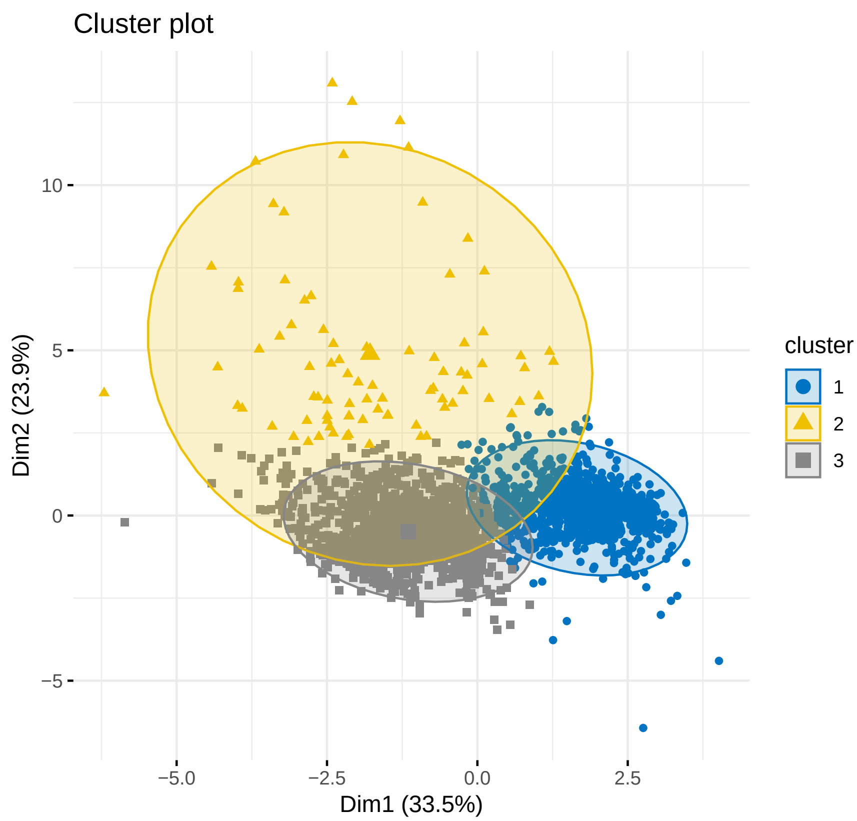


Figure 5.11: R plot of eclust showing clustering is possible using the KMeans clustering algorithm

around it. This kind of graph is expected because the distribution of the threads is probably responsible for this. In the data generation process, the threads are taken such as 10, 100, 200, 300, 400, 500, and then it is jumped to the number 1000. Now executing 200 runs for each thread, it is evident that the population for the thread distribution will be much higher in the range of 10 - 500. However, after applying the cluster results to the plotting, the divided data points are visible properly in Figure 5.13.

Clustering with KMeans requires some arguments before running the algorithm. One of the arguments is the number of clusters that we expect within the dataset. Although, we expect at least two clusters within the dataset, this argument (K = expected number of clusters) is set to 3 as our several methods for finding the optimal number of clusters indicate three clusters possible.

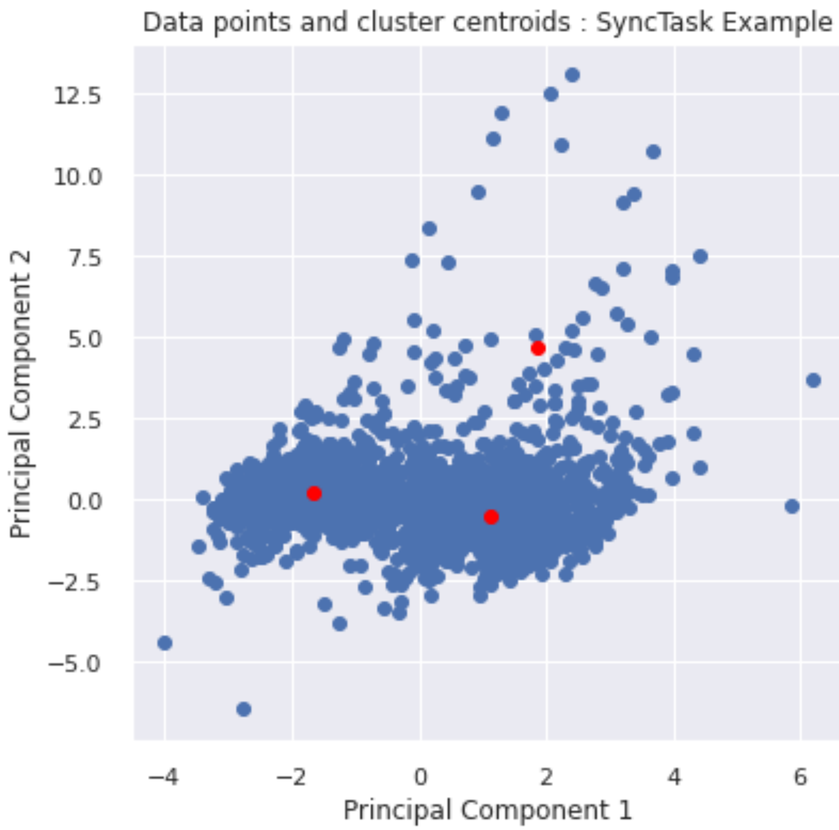


Figure 5.12: Cluster centroids after applying PCA and KMeans, extracted from lock-contention performance metrics

The argument value K is verified by the several methods including "Elbow method", "Silhouette Method" and more, which are described at Section 5.4. We maintained the following arguments for KMeans algorithm which are shown in Table 5.2:

Argument	**Value**
Expected number of clusters	3
Initialization of centroids	k-means++
Maximum Iteration	600
Number of initialization	10

Table 5.2: Required arguments that are provided to the KMeans algorithm

After KMeans, we move forward to the DBSCAN clustering to see whether the DBSCAN

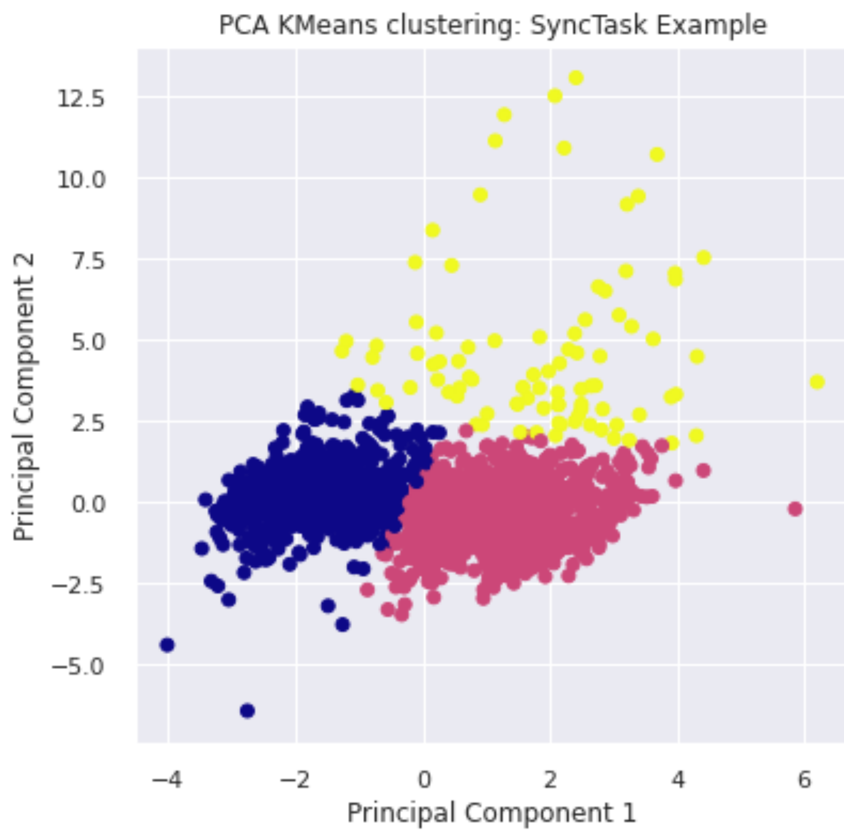


Figure 5.13: Clusters after applying PCA and KMeans, extracted from lock-contention performance metrics

clustering algorithm can classify the data from the dataset. Clustering in DBSCAN does not require the argument for the expected number of clusters to be set prior to run the algorithm. However, it requires an argument "eps" (Epsilon). Initially, we set the "eps" argument for the DBSCAN model to 0.3, classifying data into one cluster that is not expected. Tweaking the "eps" to 0.5 increases the number of groups to more than five within the dataset, also does not match our expectations. Moreover, tweaking the "eps" to a higher number does not improve the expected clustering results. The scatter plot of clustering results of DBSCAN using PCA data is shown in Figure 5.14, where it is visible that the identified clusters are not matching with our expectations. Additionally, the distribution of the data points is not properly arranged.

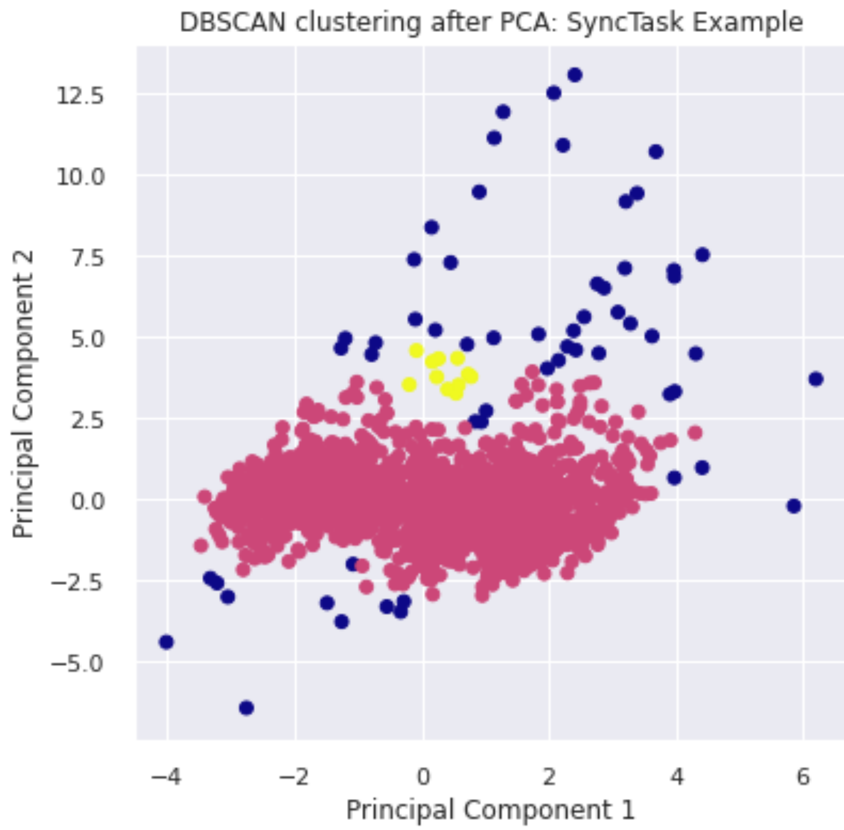


Figure 5.14: Identified clusters from PCA and DBSCAN, extracted from lock-contention performance metrics

5.6 Observing Strong Features

After the data is clustered, a reverse engineering technique is applied to determine the key features in the data for each class. We capture the clusters from the KMeans algorithm and merge the extracted clusters to the original dataset. Next, plotting the data into a radial visualization [40] gives us some insight into the strength of features in relation to each contention type class. However, we visualize this data plotting utilizing the radial visualization a little bit differently for various extracted clusters this time.

First, We merge the extracted clusters applied from PCA + KMeans to the original Python dataframe and try to visualize the dominant features for each class, as shown in Figure 5.15.

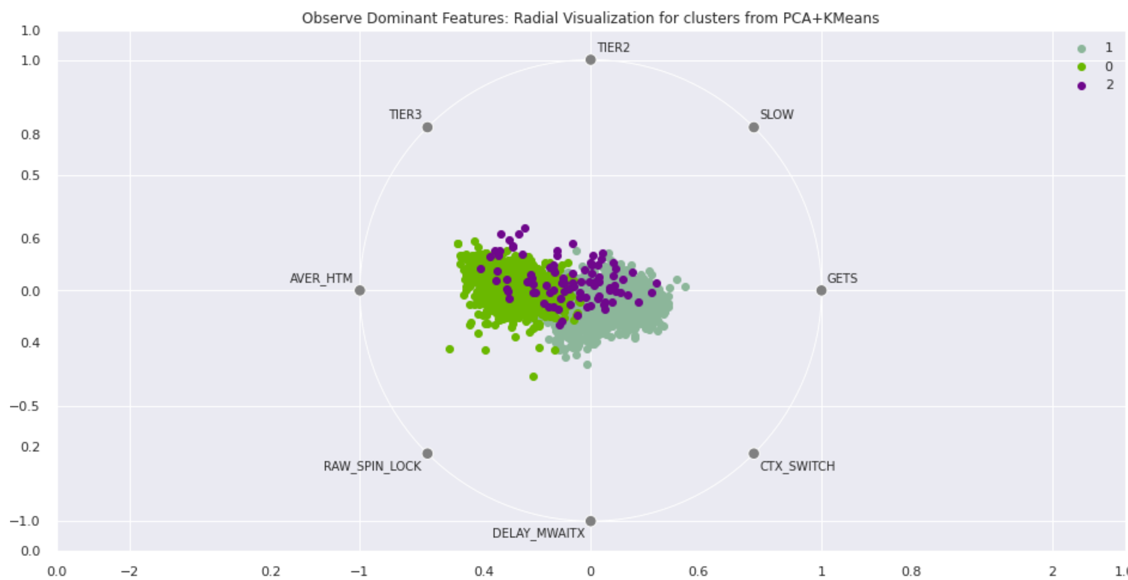


Figure 5.15: Observing key features using 2D Radial visualization for clusters extracted from PCA and KMeans.

Second, clusters extracted from KMeans only is merged to the original Python dataframe and observed the features behaviors, as shown in Figure 5.16.

Third, we go for the PCA+DBSCAN extracted clusters and observe the features, as shown in Figure 5.17.

And lastly we tried with the DBSCAN extracted clusters and observed the strong features related to each class, as shown in Figure 5.18.

We observe that KMeans algorithm performs better than The DBSCAN clustering algorithm, and it is also visible from the both Radial Visualizations (see Figures 5.17 5.18) that the clusters are not grouped together nor even they indicate proper dominant feature for each class. Hence, our analysis concludes the DBSCAN algorithm as inappropriate for our approach and the dataset.

Plotting the two Python dataframes of PCA + KMeans and only KMeans (see Figures 5.15 and 5.16) into the radial visualization does not show that many differences. Moreover, careful observation finds that they are identical. Therefore, the analysis concludes that any of the techniques can be followed or appropriate for clustering. More clearly, applying PCA before KMeans does not

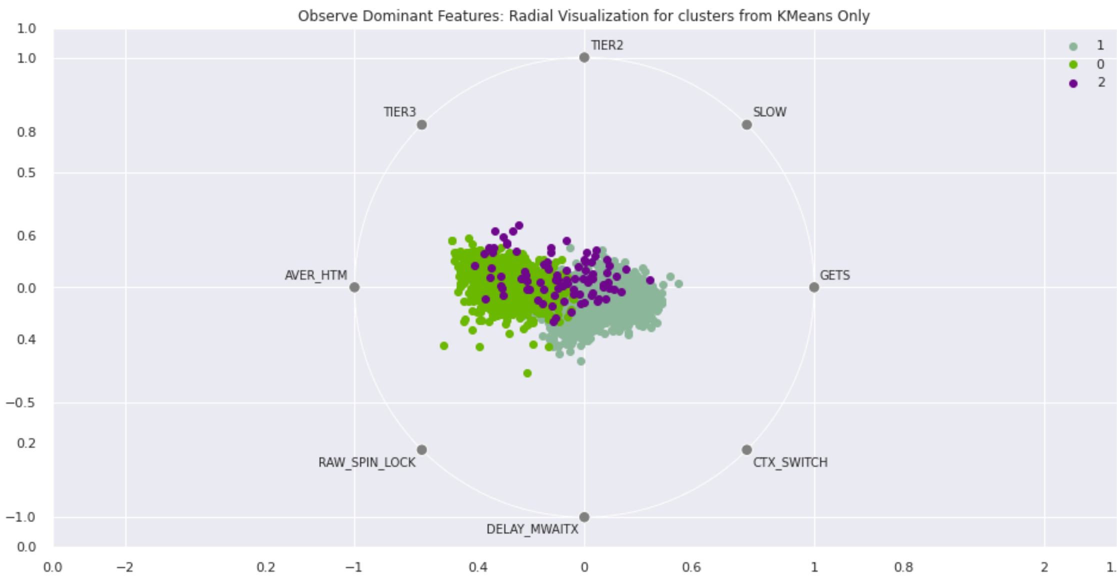


Figure 5.16: Observing key features using 2D Radial visualization for clusters extracted from KMeans algorithm only.

have that much impact on the end results of observing dominant features.

The Radial Visualization graphs consist of dataframe of PCA + KMeans / Only KMeans (see Figures 5.15 and 5.16) are able to show that some data points are leaning towards `AVER_HTM` and which refers to cluster 0 also known as (aka) fault type 1 where threads are holding the lock more than expected. However, looking at the cluster 1 and cluster 2, it is difficult to understand the strong features for them using this Radial Visualization technique. Although it is visible that cluster 2 is located towards GETS feature, that does not finalize whether this cluster belong to fault type 2.

However, after changing the orientation of the features around the circle of the radial visualization, it shows some distinction among the clusters and the dominant features. The former orientation of the features around the circle of the radial visualization was GETS $>$ CTX_SWITCH $>$ DELAY_MWAITX $>$ RAW_SPIN_LOCK $>$ AVER_HTM $>$ TIER3 $>$ TIER2 $>$ SLOW, placing the features clockwise starting from GETS. This previous orientation experiences the clusters overlapping. Hence a new orientation of the features is recommended to see whether it can separate the clusters well. In this new design, the features are plotted with the following orientation: GETS $>$

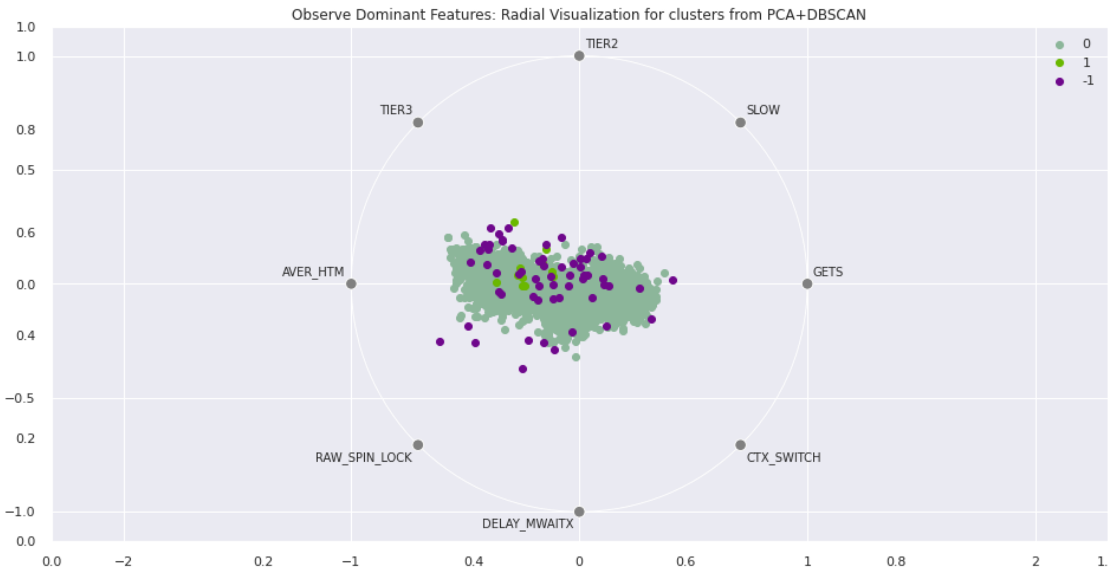


Figure 5.17: Observing key features using 2D Radial visualization for clusters extracted from PCA and DBSCAN algorithms.

RAW_SPIN_LOCK > TIER3 > TIER2 > CTX_SWITCH > AVER_HTM > DELAY_MWAITX > SLOW. The new orientation-based radial visualization plot for PCA+KMeans extracted clusters is shown in Figure 5.19. Although it is seen from the figure that some data points are inclined towards AVER_HTM, some are towards GETS and some are TIER2 and TIER3, due to large overlapping among the clusters it is difficult to conclude if there are dominant features between the two clusters.

5.7 Model's Performance Evaluation

At this moment, we generate a synthetic dataset by emulating the whole process using example concurrent codes. Although the whole process is an emulation, our generated data shows some distinct classes on which we execute some model's performance evaluations. The clustering approach using PCA+DBSCAN and DBSCAN only fails to show expected performance by producing unexpected classes, and hence we left them out from the performance evaluation. We perform two sets of validation, one for the technique where the clustering process is performed using PCA

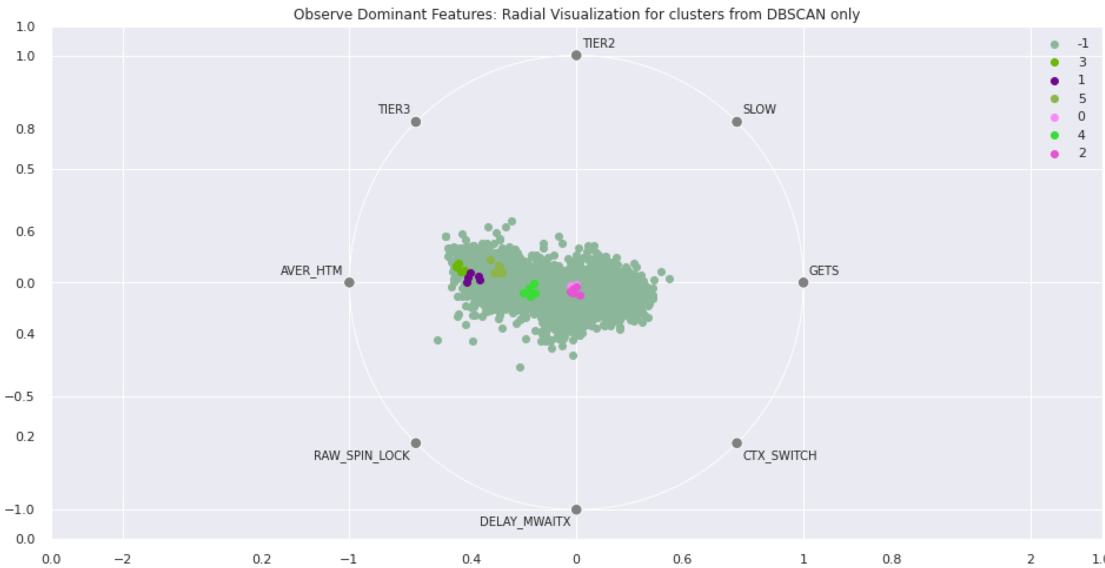


Figure 5.18: Observing key features using 2D Radial visualization for clusters extracted from DBSCAN algorithm only.

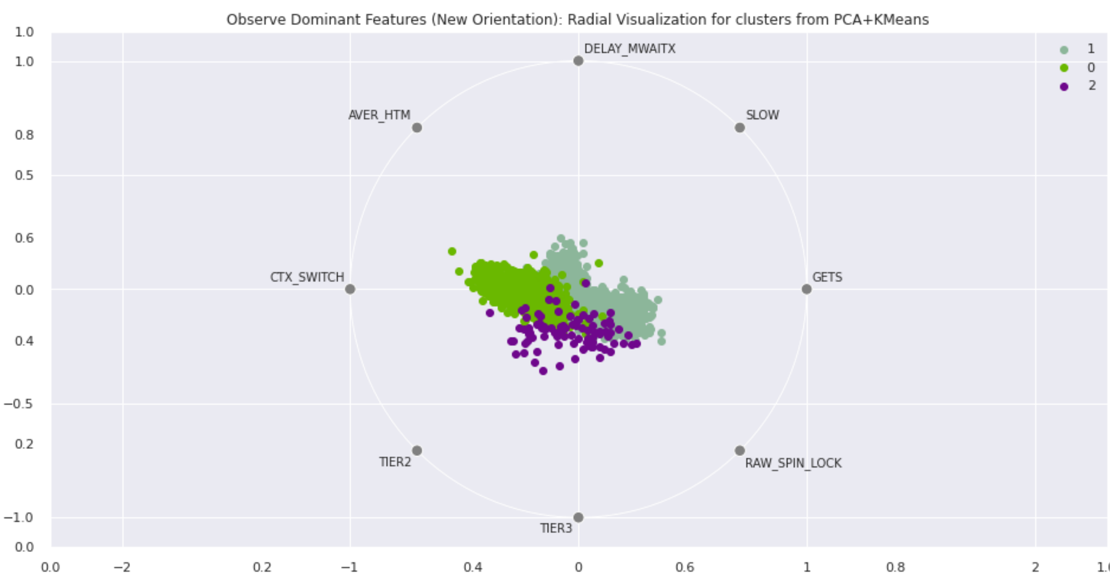


Figure 5.19: Observing key features using 2D Radial visualization for clusters extracted from PCA+KMeans algorithm (New orientation).

+ KMeans, and the other one is using KMeans only. However, running the performance evaluation approach "Training Test Split" results in the accuracy of 93.61%. In order to perform this split into training test validation we utilize the method `train_test_split` from the library `sklearn.model_selection`. The other arguments we use for this validation are:

- **"Training Test Split" Validation:**
  - Test size: 0.25 (25%)
  - Random state: 7 (randomly chosen)

As we have a label for the dataset, we fit our data leveraging widely used Logistic Regression model [41] from the library `sklearn.linear_model`. After that, the method `accuracy_score` from the library `sklearn` helps us determine the prediction's accuracy score. In our second validation for the same PCA+KMeans dataframe, the technique called K-fold cross validation is utilized. The arguments for the validation are:

- **"K-fold Cross" Validation:**
  - Number splits: 10 (10-fold cross validation)
  - Random state: 7

However, we used the same Logistic Regression model, and the validation resulted in the accuracy of 94.32% (Performance mean) and 1.80% (Performance deviation). This technique leverages the method `cross_validation_score` from the Python library, `sklearn.model_selection`. "Training Test Split" performance evaluation with the cluster extracted from KMeans only gives us the accuracy of 96.16%. In contrast, performance evaluation with the method "K-fold cross" performance evaluation gives us the accuracy of 95.91% (performance mean) and 1.36% (performance deviation). The performance results are listed in Table 5.3. Observing the final performance evaluation, it can be concluded that using just KMeans yields better accuracy than the performance of PCA+KMeans.

Dataframe	Training Test Split	K-fold Cross Validation
KMeans+PCA	93.61%	94.32% (mean), 1.80% (deviation)
KMeans	96.16%	95.91% (mean), 1.36% (deviation)

Table 5.3: Models' performance evaluation on different dataframe

5.8 SUMMARY

In this chapter, we try to present all kinds of preprocessing results and validations before the clustering and the final clustering results. However, final clustering results show that three clusters are possible within the dataset we generated, and this is also verified by some methods we use for clustering assessment. Preprocessing starts from heatmap analysis, where we manage to show that some features are highly correlated and some are not. Based on this heatmap analysis, some features are filtered. After that, a series of clustering assessments show that the optimal number of clusters possible within the dataset is three. Scaling is mandatory for clustering techniques, and after ensuring scaling, the dataset dimension is reduced to only two components (PC1 and PC2) by applying principal component analysis. We then feed the KMeans with processed PCA data, which confirms three clusters. However, we initially try to plot the resulting clusters to a radial visualization to observe strong features for each cluster. Although radial visualization partially successful to show the dominant features for all clusters, dominant feature for cluster 0 (High hold time fault) can be observed through this technique only. Additionally, changing the orientation of the features around the circle successfully separates those clusters.

Chapter 6

CLUSTER ANALYSIS

6.1 INTRODUCTION

Extracted clusters from KMeans have numerical labels, and these do not help us identify the actual label (e.g., Fault-1, Fault-2, etc.) for each data point. Therefore, a method is required that will assist us in labeling each data point to its actual fault type, which is called semantic labeling. In this chapter, we try to present a procedure to label them by observing the distribution of the input parameters, (e.g., THREADS and SLEEP) that we collect during the code execution.

Although Radial Visualization with new orientation of the features assists us in showing strong features for the three clusters, it is difficult to identify the dominant features for the clusters precisely as the overlapping among the clusters are high. Therefore we move to a different visualization technique to understand the dominant features for each cluster. In order to achieve that, we first apply necessary clustering algorithms and obtain the clusters, then we merge the cluster results to the original dataset keeping the data-frame's index unchanged. After that, we plot each feature in a box plot to observe the value distribution for them. Therefore, the box plots will help to reveal the dominant features for each of the clusters. In this chapter, our primary target is to finalize our hypothesis that our assumption regarding the fault types is correct. These faults are reflected in the run-time logs from where those can be detected leveraging our clustering approach. We try to plot each feature in a box plot and observe their distribution for the different clusters to see which features are dominant for a particular cluster.

6.2 Labeling the Clusters

Before classifying each data point by fault-type a preprocessing algorithm is applied. It requires the test parameters such as the number of threads and sleep times, which we vary during our code execution. We store those parameters information for each run and map back to the original dataset after a successful clustering process. Our belief is, plotting these number of threads and sleep time should assist us in finding the original fault label for each data point.

During the Threads-Cluster box plot observation and labeling the clusters, the parameter "THREADS" is mapped back to the scaled final dataset on which we perform clustering. However, the column "THREADS" is not scaled and we add the unchanged original value stored during the dataset generation. We follow the same exact procedure for the parameter "SLEEP" when it is added to the scaled dataset.

According to the hypothesis, the thread number is one of the main differences between the regular contention and contention fault type 2, which is a high-frequency requests problem. As the fault type 2 problem depicts itself that too many threads send access requests to the locked resources, hence in the threads distribution, the cluster representing fault type 2 should gain a high number of thread values compared to the other contention clusters. After plotting the threads distribution in a box plot for each cluster, it is visible that one of the clusters contains the threads distribution with a higher number of threads. It proves our hypothesis that there is a relation between fault type 2 and the thread numbers where the thread numbers are high in values. The thread distribution box plot is shown in Figure 6.1. Based on the figure it implies that cluster 2 may fall under fault type 2, where request frequencies from the threads are too high.

In order to prove our following hypothesis that there is a relation between the high hold-time fault type and sleep time (execution time) where the sleep times are increased in values, we plot the

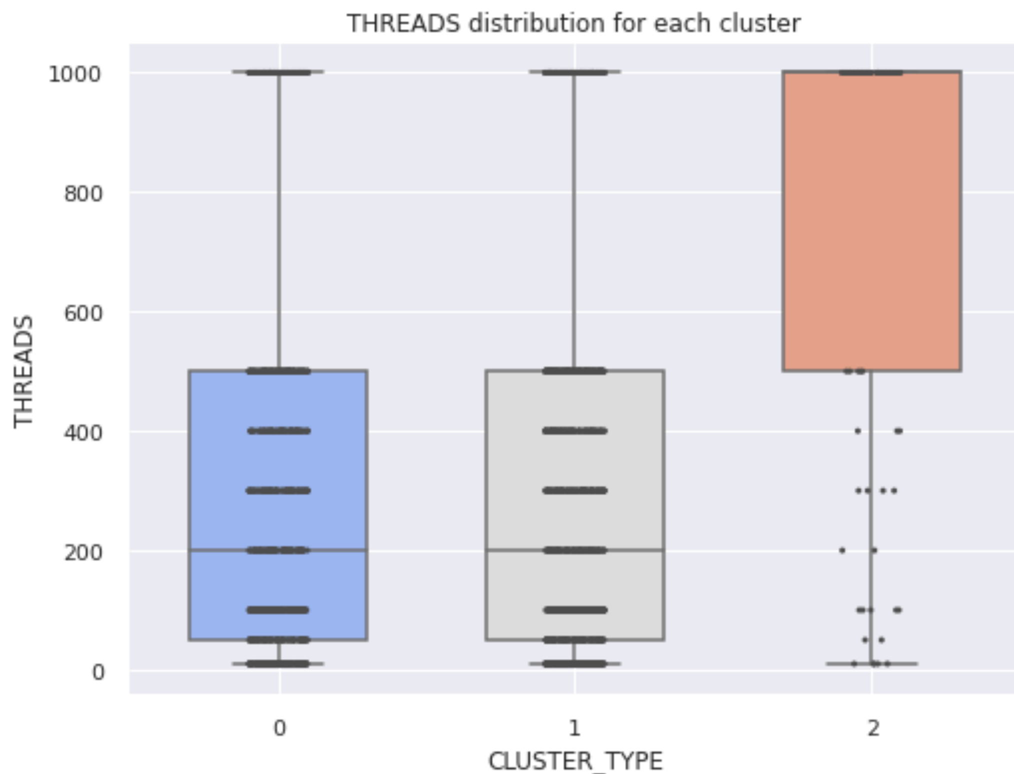


Figure 6.1: Observing Threads distribution using box plot visualization for each cluster

sleep times for each cluster to a box plot. The box plot of sleep times shows the distribution, which is shown in Figure 6.2. The figure clearly illustrates the situation that CLUSTER_TYPE 0, which is the fault type 1 (high hold-time), contains the sleep times distribution higher than the other two clusters. Moreover, from Figure 6.2 it is also visible that CLUSTER_TYPE 1 carries the sleep times distribution, which is lower compared to the other two clusters, representing the low contention cluster.

Although we expect two clusters from our dataset, either cluster representing high hold-time fault or high-frequency requests fault, methods for validating the optimal number of clusters show three. Now, we assume CLUSTER_TYPE 1 in our dataset probably represents the low contention cluster as the sleep times distribution is lower compared to the other two. It also confirms that the low contention cluster displays a lower sleep times distribution than both fault type 1 and fault type

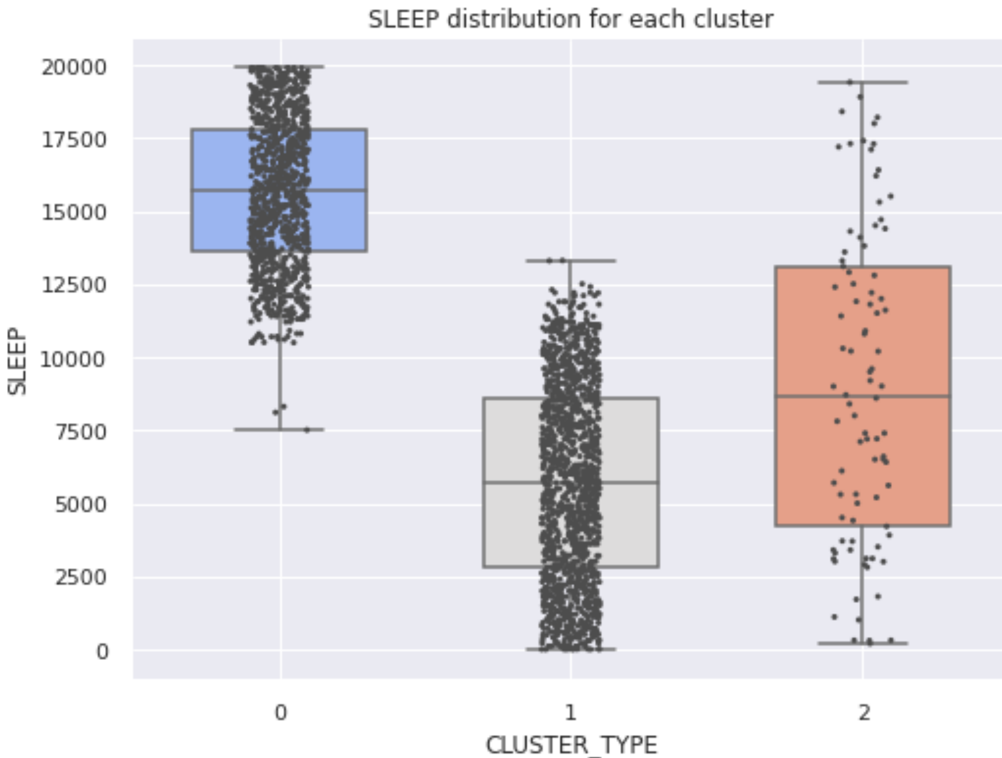


Figure 6.2: Observing Sleep distribution using box plot visualization for each cluster

2.

6.3 Observing Features : An Advanced Analysis

Observing the features and exploring dominant features for each cluster is always recommended for our work, as our investigation regarding strong features exploring is still in progress. In order to explore dominant features for each class, we take the assistance of this box plot visualization method as it presents the distribution of a particular feature against the clusters. The features' values that we plot against the clusters are scaled values, and they are scaled leveraging Python package **Standard Scaler** at the time of clustering. As the data is considered from the final dataset and those data points are scaled and unchanged, box plots show scaled features against clusters. The different plotting of the features are listed below:

6.3.1 Observing GETS:

The GETS feature of JLM represents the total number of successful lock acquisitions. Therefore, it is pretty straightforward that the low contention cluster should gain the high range of GETS values compared to the other two clusters, and it is also visible in Figure 6.3. However, from the figure it is also visible that the GETS distribution for cluster 2 overlaps with cluster 1. This scenario implies that the metrics related to lock acquisition increase unless the threads stay inside the critical section for too long. The GETS distribution figure proves that the high hold time cluster negatively correlates to the GETS value. If the lock is acquired and held for a long time, then the other threads wait to obtain it, as a result, acquisition decreases.

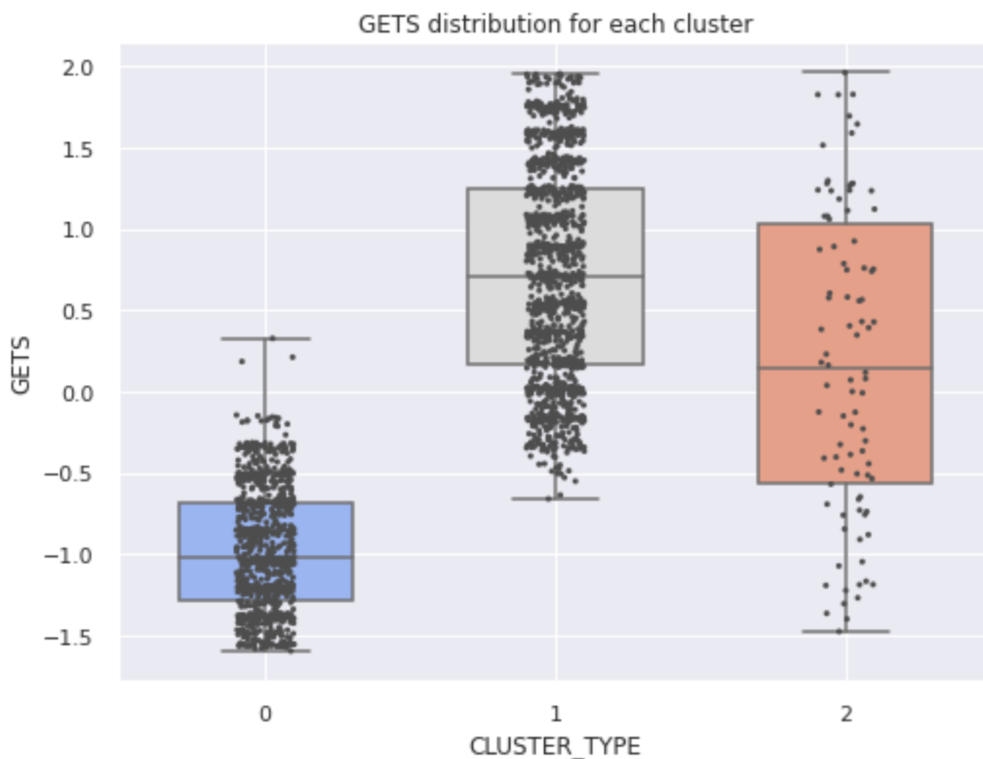


Figure 6.3: Observing GETS feature distribution using box plot visualization for each cluster

6.3.2 Observing Spin Features:

Expectation from the features related to spin count (e.g., TIER2, TIER3, _raw_spin_lock) is that they should experience high numbers when multiple threads send requests simultaneously to the locked resources. Therefore, based on the experiment, fault type 2 should possess high values in features related to spin counts. Plotting these spin counts in box plots reveals that cluster 2 has a high range of distribution both for TIER2 and TIER3 indicating the high-frequency access requests bottleneck. Hence, these TIER2 and TIER3 counts can be the distinguishing factors for fault type 2 performance issue. Both box plot of TIER2 and TIER3 are shown in Figure 6.4 and Figure 6.5.

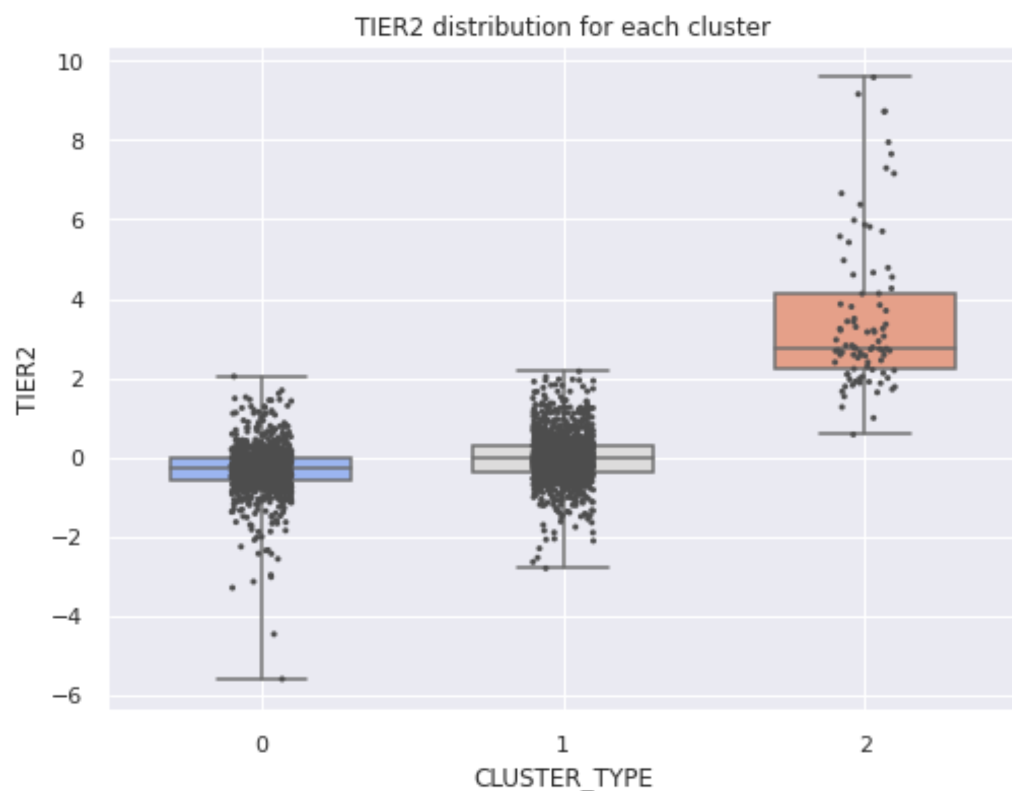


Figure 6.4: Observing TIER2 feature distribution using box plot visualization for each cluster

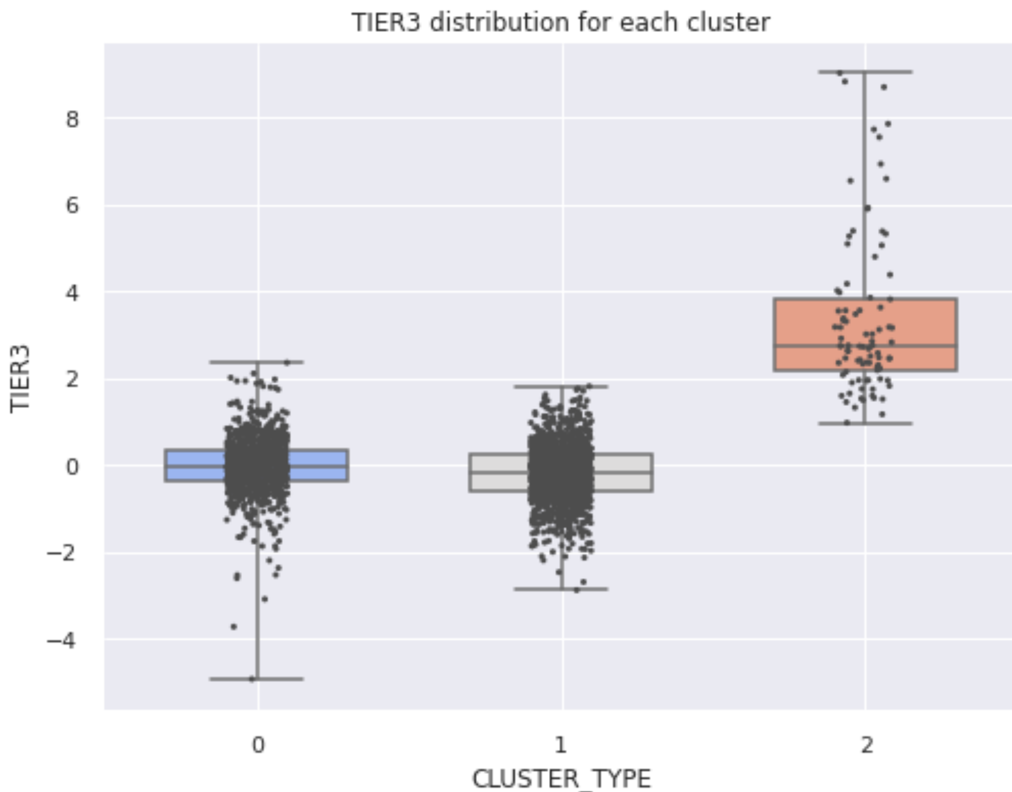


Figure 6.5: Observing TIER3 feature distribution using box plot visualization for each cluster

6.3.3 Observing AVER_HTM:

JLM feature AVER_HTM represents the average hold-time and negatively correlates to the GETS value we obtained from heatmap analysis. Hypothetically, when contention occurs due to a high hold-time performance issue, this AVER_HTM feature increases in number. Hence, a cluster representing high hold-time should obtain an increased range of AVER_HTM distribution. The box plot of AVER_HTM vs. clusters plotting shows our expected results, and it can be observed in the AVER_HTM distribution box plot as shown in Figure 6.6. Moreover, the Figure also illustrates that the low contention cluster gains a lower range of AVER_HTM distribution, and fault type 2 is relatively higher than the low contention.

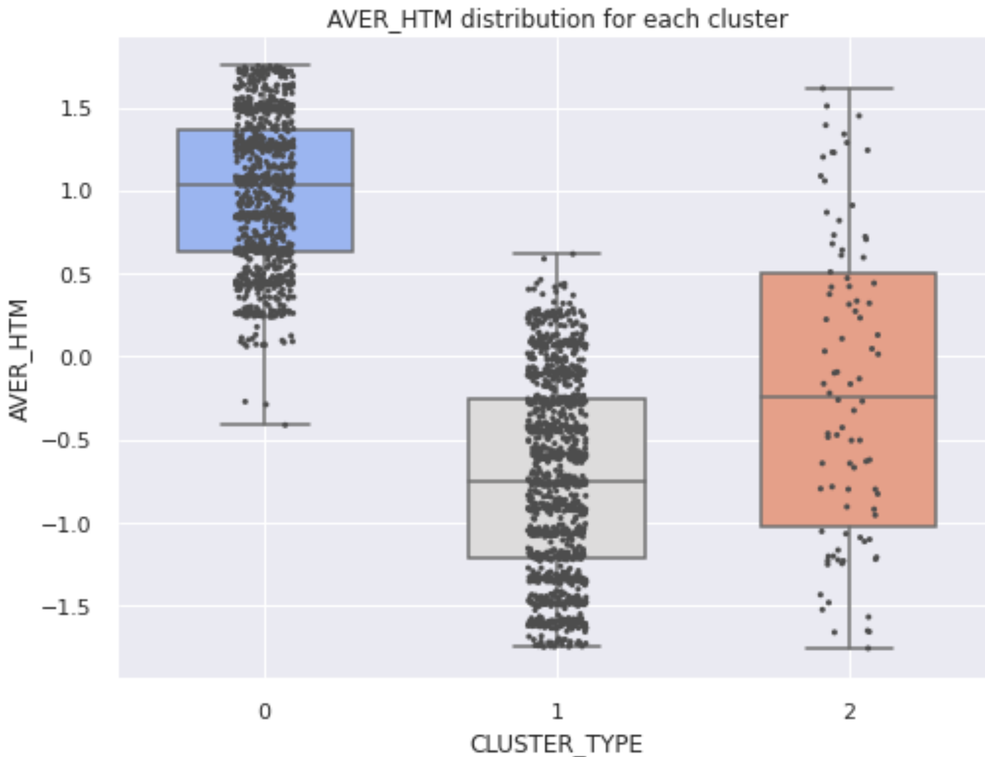


Figure 6.6: Observing AVER_HTM feature distribution using box plot visualization for each cluster

6.3.4 Observing The Other Features:

Although plotting the other features from the *perf* data such as RAW_SPIN_LOCK, CTX_SWITCH and DELAY_MWAITX do not assist us in distinguishing the fault types explicitly. Their distribution overlap more, but the feature RAW_SPIN_LOCK is positively correlated to spin-related features. Hence the cluster 2 (frequent access fault) has relatively higher counts than the other two. The box plot of RAW_SPIN_LOCK is shown in Figure 6.7. Kernel symbol CTX_SWITCH appears more when there are more threads compete each other to acquire the lock and therefore it slightly increases for cluster 2 again. See the box plot for CTX_SWITCH in Figure 6.8. However, the feature DELAY_MWAITX is kernel symbol represents monitor wait sample counts, showing less range in here for high hold time faults but during experiment our analysis observe that it increases tremendously when critical section is held for more than equal to one millisecond. The feature SLOW fails to present any interesting characteristics in our experiment. The box plot of features DELAY_MWAITX and SLOW are shown in Figure 6.9 and Figure 6.10 respectively.

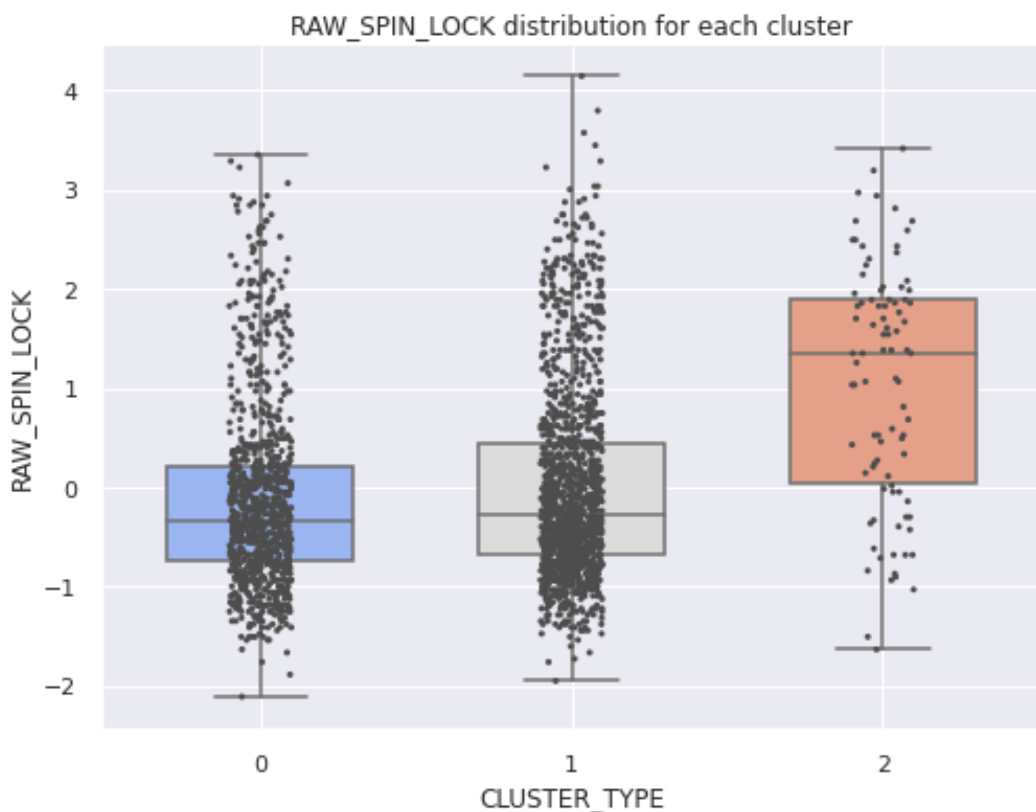


Figure 6.7: Observing RAW_SPIN_LOCK feature distribution using box plot visualization for each cluster

After analyzing the plotting of all of those graphs we can conclude the followings:

- “Less Contention” has the low spinning counts as well as low hold times but the lock acquisition is higher.
- “Contention Fault 1” has low spinning counts but high in hold times.
- “Contention Fault 2” has high spinning counts but low in hold times.

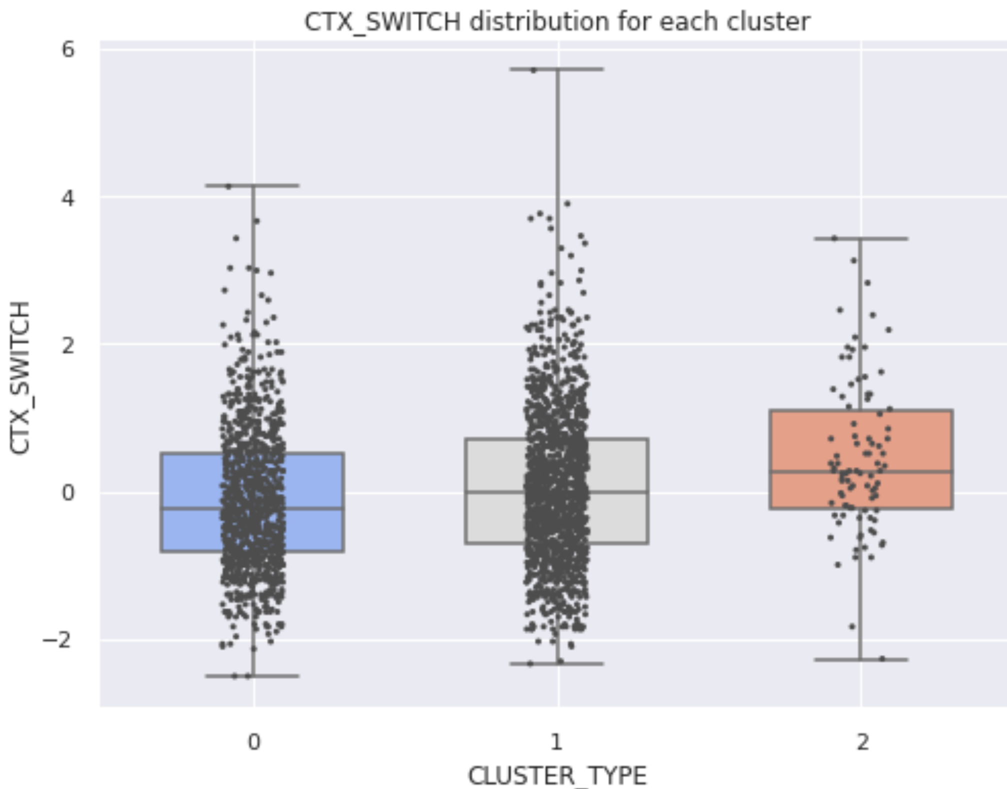


Figure 6.8: Observing CTX_SWITCH feature distribution using box plot visualization for each cluster

6.4 Dominant Features : Importance to The Developers

It is important to know why the dominant features are crucial to the developers and performance engineers. A clustering technique helps classify some groups out of the dataset where the actual labels for the groups are unknown. In order to label these clusters, the dominant features play a significant role. A data point with a high hold-time falls under fault type 1. Again if the data point obtains high values on spin-related counts (TIER2 and TIER3), it falls under fault type 2. In this case, A threshold value is needed to consider the high hold time or high spin counts. In order to obtain and reveal the threshold value of high hold-time or high spin counts, a decision tree model can be trained using the KMeans-extracted labeled data. Once the decision tree produces the threshold

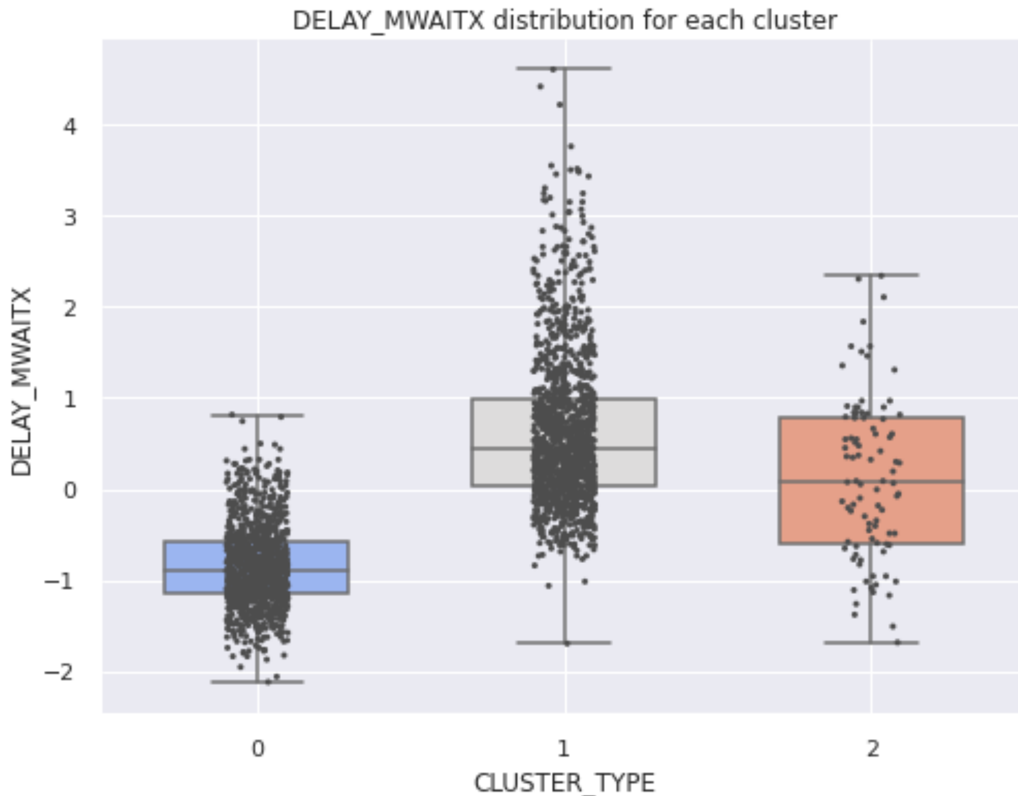


Figure 6.9: Observing DELAY_MWAITX feature distribution using box plot visualization for each cluster

value, a new data point can be compared and given its actual label.

Based on the labeling, it is also possible to generate recommendations. In case of a high hold-time situation, our approach should recommend that the lock consume more than expected time, and the time can be reduced by optimizing some unnecessary computations inside. On the other hand, in case of high-frequency access by the threads, our approach should notify about reducing lock access by separating the shared resources into multiple locks (lock splitting) or making the shared resources more granular during reading and writing.

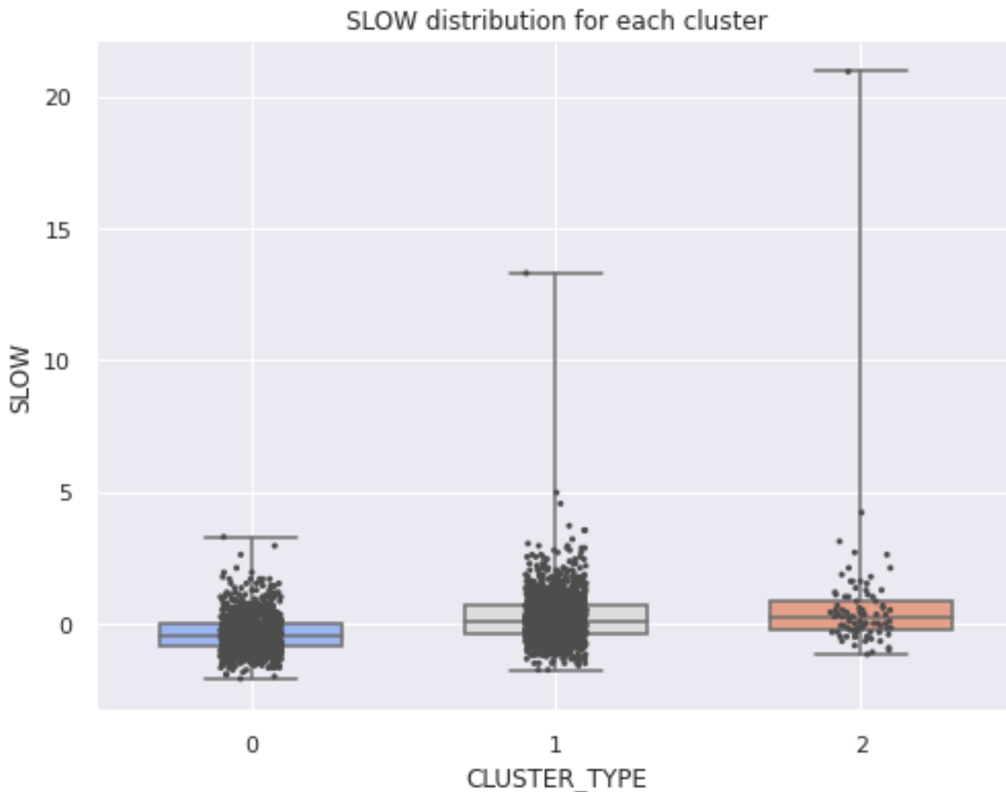


Figure 6.10: Observing SLOW feature distribution using box plot visualization for each cluster

6.5 SUMMARY

When radial visualization partially successful to reveal the crucial features, an advanced analysis with box plotting comes with great help to understand the main features for each of the clusters. In this chapter, we try to present that some features are dominant for each cluster, such as, when it is a high hold-time issue, then AVER_HTM will be higher than any other clusters present in the dataset. When AVER_HTM feature increases, then lock acquisition metrics (e.g., GETS) decrease in number. Features related to spin counts (e.g., TIER2, TIER3) are dominant features for high-frequency access problem. Lastly, we manage to show that the low contention cluster contains high lock acquisition counts, which is represented by the GETS feature. These features for each cluster are essential to the developers and the performance engineers to understand the actual issue and what type of solution they should apply to reduce the contention.

Chapter 7

CONCLUSIONS

7.1 OVERVIEW

In this research we try to prove a hypothesis that lock contention fault types can be classified through run-time traces via the training of an unsupervised classifier. It is possible, because the fault types produce some patterns in the run-time performance metrics when different types of contention occurs. Initially we studied the java intrinsic lock and the locking mechanism. Our empirical study shows that, according to Goetz, lock contention performance bottlenecks are primarily caused by two reasons. First, threads hold the critical section for longer than expected, and second, threads send access requests to the critical section with high frequency. Issues with these bottlenecks cannot be expressed as bug, because bug produces faulty results whereas performance bottlenecks reduce application performance or in other words, reduce application throughput. Later our study moved forward with learning more about the tool named JLM that IBM uses for their internal use to profile java applications' health. We carefully studied the JLM data and the java inflated monitors which was essential for our study. Understanding the JLM tool and its log accelerated our work. Not only JLM but the perf trace is also equally essential to identify the lock contention faults and classify them. Our research showed that when contention faults occur the perf trace experiences with some common symbols and our analysis carefully studied them as well.

However, our experiment started with creating example code to emulate lock contention faults. We executed our code in a controlled environment so that we can control threads and sleep time inside the critical section emulating different execution times.

We built a parser that parses the JLM and perf data from the raw JLM and perf data. After collecting the data from both JLM and perf we merged them into single file and applied heatmap to perform the initial data preprocessing that helps us reducing some less necessary features from the dataset. At our final stage we applied Principal Component Analysis (PCA) to reduce the dimentionality into two final dimensions. We applied the KMeans using the PCA extracted dataframe and found that the data forms some clusters based on the dominant behaviors of the features. Clusters showed that one group, fault type one, spent too much time inside the critical section, the second group is low contention, and the last one contains the high spin counts represents the fault type two where increased number of threads make requests to the locked resources with high frequency.

Later on we evaluated our model using "raining test split" and "k-fold cross validation" methods and they show an accuracy of approximately 94%.

At the end of our thesis, some advanced analyses are performed to label the fault types and observe the dominant features for each cluster. Leveraging the box plot and plotting the threads and sleep times along with the clusters reveals that contention due to high hold time is related to high sleep times, and fault due to high-frequency access is related to an increased number of threads. When radial visualization partially reveals the important features for each cluster, box-plot comes with great help to understand some dominant features for the clusters. However, our final analysis concludes that the features are essential to the developers that can be utilized to solve the area of a codebase responsible for contention bottlenecks. Based on these features, some suggestions can be thrown to apply some solutions to the faulty region of the codebase.

7.2 Answer to Those Research Questions

At the beginning of the thesis some research questions are highlighted. This research intend to classify lock contention fault types and deliver a method that helps identify faults with ease. However, our study tries to answer those primary questions regarding contention classification using the clustering ML approach.

1. **How is this method good enough over traditional approaches?** To answer this question, it is worth mentioning an approach that IBM's performance engineers follow. The "IBM Performance Inspector" is a monitoring tool that helps performance engineers to identify and locate performance-related faults due to locking in a java application. Although this tool delivers its job quite well, it needs some manual intervention and manual analysis. To detect locking bottlenecks, engineers follow the steps below:

   Let us consider an application that creates a locking problem, but we do not know if it is related to contention. The first step would be to check the *perf* profile and then the JLM stat.

   - **STEP 1 (Check *perf* profile)**:
     - Run *perf* record and check the profile data.
     - Check how much time is spent on which known routines related locking issue.
     - If the locking related routines are prominent, then it is worth spending time to debug JLM data.
   - **STEP 2 (Check JLM stat)**:
     - Check for java application-related monitors under "Java Inflated Monitors" section.
     - Check if the AVER_HTM is high in counts, it is assumed that probably the problem indicates holding the critical section longer than expected.

After identifying the issue, they usually go back to the codebase and search for the possible locks responsible for the contention and performance degradation issue. Next, the engineers try to resolve the problem based on the JLM metrics. Although performance engineers are capable of solving issues related to high hold time, this approach fails to provide necessary instructions to deal with performance degradation due to high-frequency access issues. Moreover, our method reduces additional efforts and human intervention that performance engineers put into identifying these contention faults.

2. **Why ML is needed for this type of work?** Lock contention performance-related data is numeric and such high range of numbers are often impossible to digest easily by the engineers. In our opinion, ML approach helps to visualize the fault types and translate and transfer the necessary instructions to the developers.

7.3 Limitations of our approach

The very first limitation of our approach is the lack of a proper dataset. Due to the lack of an available dataset, it is difficult to train and then run an unsupervised ML algorithm and acquire the desired clusters of fault types. We tried to generate some synthetic data by running an example concurrent code. However, we do believe our dataset can be extended by executing some concurrent example codes with faults in them. It needs exploration of example codes in open source repositories such as GitHub. Therefore, another problem we experienced is the lack of concurrent example codes. Moreover, there are not so many real-world java applications with faults; we can use them as benchmark applications.

Our second limitation of the approach is the run-time data we collect which might be in some cases incompatible with other types of operating systems. Our approach collects logs from the kernel, which is Linux-based, and some operating systems do not share the same kernel. Therefore collecting features could be a problem in other types of OS(s).

And finally, the last limitation of our approach is the JVM we choose. JLM is compatible with OpenJ9 and incompatible with other JVM(s) such as HotSpot. Therefore, our approach might be vulnerable to these situations. However, even though we continued our experiment with synthetic data, from the JLM and *perf* performance data and observing their behavior, it can be assumed that the faults can be classified to help the developers with proper recommendations.

7.4 Future work

Our plan is to collect concurrent codes with faults as many as we can in the future. By executing the concurrent codes, we will collect necessary JLM and *perf* data and create a proper dataset. Therefore it can be used as an iconic dataset for identifying or classifying contention-related faults. Moreover, we believe, through our research, it is also possible to extract some other types of faults that are currently unknown. Therefore, our research has another potential work to label the different fault types, and we also have a plan for that. Additionally, we will try to collect real-world example java applications with faults in them so that those can be used for benchmark as well as performance evaluation. We strongly believe our final training corpus will significantly contribute to the research community who work with contention-related fault identification and classification process through the ML approach.

Bibliography

[1] Oracle, *Intrinsic Locks and Synchronization (The Java™ Tutorials > Essential Classes > Concurrency)*, 2015. [Online]. Available: `https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html` (visited on 09/09/2021).

[2] T. Locks, "FeatherWeight Synchronization for Java, David F, Bacon et al," in *Proceeding of the ACM COnference on Programming Language Design and Implementation, SIGPLAN*, 1998, pp. 258–268.

[3] B. Göetz and A. W. Professional, "Java Concurrency In Practice," *Building*, vol. 39, no. 11, p. 384, 2006, ISSN: 03008495. [Online]. Available: `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Java+Concurrency+in+Practice#0`.

[4] T. Yu and M. Pradel, "Pinpointing and repairing performance bottlenecks in concurrent programs," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3034–3071, 2018, ISSN: 15737616. DOI: `10.1007/s10664-017-9578-1`.

[5] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," *2015 10th Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings, pp. 426–438, 2015. DOI: 10.1145/2786805.2786815.

[6] M. Kohler, *A Simple Way to Analyze Thread Contention Problems in Java*, 2006. [Online]. Available: https://blogs.sap.com/2006/10/18/a-simple-way-to-analyze-thread-contention-problems-in-java/ (visited on 12/08/2021).

[7] N. Salnikov-Tarnovski, *Improving Lock Performance*, 2015. [Online]. Available: https://dzone.com/articles/improving-lock-performance (visited on 12/08/2021).

[8] IBM, *Java Lock Monitor*. [Online]. Available: http://perfinsp.sourceforge.net/examples.html#jlm (visited on 10/18/2021).

[9] YourKit, *YourKit Java Profiler*. [Online]. Available: https://www.yourkit.com/java/profiler/features/ (visited on 09/09/2021).

[10] E.-t. GmbH, *The Award-Winning All-in-One Java Profiler*, 2001. [Online]. Available: https://www.ej-technologies.com/products/jprofiler/overview.html (visited on 11/11/2021).

[11] Kernel.org, *Linux kernel profiling with perf*, 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial (visited on 10/16/2021).

[12] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the Free-Lunch profiler," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pp. 291–307, 2014. DOI: 10.1145/2660193.2660210.

[13] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings international parallel and distributed processing symposium*, IEEE, 2003, 7–pp.

[14] C. Zhang, J. Li, D. Li, and X. Lu, "Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems," *IEEE Access*, vol. 7, pp. 99 123–99 135, 2019, ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2923956.

[15] I. Corporation, *IBM Monitoring and Diagnostic Tools - Health Center*, 2007. [Online]. Available: https://www.ibm.com/docs/en/mon-diag-tools?topic=monitoring-diagnostic-tools-health-center (visited on 10/09/2021).

[16] Oracle Corporation, *Visual VM*. [Online]. Available: http://visualvm.java.net/ (visited on 11/06/2021).

[17] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 269–279, 2010, ISSN: 15232867. DOI: 10.1145/1837853.1693489.

[18] P. Hofer, D. Gnedt, A. Schörgenhumer, and H. Mössenböck, "Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level," *ICPE 2016 - Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering*, pp. 263–274, 2016. DOI: 10.1145/2851553.2851559.

[19] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 245–254, 2010, ISSN: 02705257. DOI: 10.1145/1806799.1806838.

[20] R. Gopalakrishnan, P. Sharma, M. Mirakhorli, and M. Galster, "Can latent topics in source code predict missing architectural tactics?" In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 15–26.

[21] J. Hale, *Scale, Standardize, or Normalize with Scikit-Learn*, 2019. [Online]. Available: https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02 (visited on 12/07/2021).

[22] K. Ajitesh, *Correlation Concepts, Matrix & Heatmap using Seaborn*, 2020. [Online]. Available: https://vitalflux.com/correlation-heatmap-with-seaborn-pandas/ (visited on 06/21/2021).

[23] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.

[24] I. T. Jollife and J. Cadima, "Principal component analysis: A review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20 150 202, 2016, ISSN: 1364503X. DOI: 10.1098/rsta.2015.0202.

[25] M. Galarnyk, *PCA using Python (scikit-learn)*, 2017. [Online]. Available: https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60 (visited on 06/21/2021).

[26] M. Daszykowski and B. Walczak, "Density-Based Clustering Methods," *Comprehensive Chemometrics*, vol. 2, pp. 635–654, 2009. DOI: `10.1016/B978-044452701-1.00067-3`.

[27] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN," *ACM Transactions on Database Systems*, vol. 42, no. 3, pp. 1–21, 2017, ISSN: 15574644. DOI: `10.1145/3068335`.

[28] S.-y. Developers, *RadViz Visualizer*, 2016. [Online]. Available: `https://www.scikit-yb.org/en/latest/api/features/radviz.html` (visited on 10/24/2021).

[29] K. Academy, *Box Plot Review*, 2021. [Online]. Available: `https://www.khanacademy.org/math/statistics-probability/summarizing-quantitative-data/box-whisker-plots/a/box-plot-review` (visited on 11/24/2021).

[30] M. Galarnyk, *Understanding Box Plots*, 2018. [Online]. Available: `https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51` (visited on 11/24/2021).

[31] I. Eclipse Foundation, *OpenJ9*. [Online]. Available: `https://www.eclipse.org/openj9/` (visited on 10/16/2021).

[32] S. Chatterjee, "Good Data and Machine Learning," 2017. [Online]. Available: `https://towardsdatascience.com/data-correlation-can-make-or-break-your-machine-learning-project-82ee11039cc9`.

[33] A. Kassambara, *Cluster Validation Essentialss*, 2018. [Online]. Available: `https://www.datanovia.com/en/lessons/assessing-clustering-tendency/` (visited on 11/17/2021).

[34] J. S. Franklin, *Elbow method of K-means clustering using Python - Analytics Vidhya - Medium*, 2019. [Online]. Available: `https://medium.com/analytics-vidhya/elbow-method-of-k-means-clustering-algorithm-a0c916adc540` (visited on 06/21/2021).

[35] H. B. Zhou and J. T. Gao, "Automatic method for determining cluster number based on silhouette coefficient," in *Advanced Materials Research*, ser. Advanced Materials Research, vol. 951, Switzerland: Trans Tech Publications Ltd, 2014, pp. 227–230, ISBN: 9783038351320. DOI: `10.4028/www.scientific.net/AMR.951.227`.

[36] K. Arvai, *Knee-point Detection in Python*, 2019. [Online]. Available: `https://github.com/arvkevi/kneed` (visited on 10/20/2021).

[37] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a" kneedle" in a haystack: Detecting knee points in system behavior," in *2011 31st international conference on distributed computing systems workshops*, IEEE, 2011, pp. 166–171.

[38] K. Arvai, *K-Means Clustering in Python: A Practical Guide*. [Online]. Available: `https://realpython.com/k-means-clustering-python/#hierarchical-clustering` (visited on 10/06/2021).

[39] M. Charrad, N. Ghazzali, V. Boiteau, and A. Niknafs, “Nbclust: An R package for determining the relevant number of clusters in a data set,” *Journal of Statistical Software*, vol. 61, no. 6, pp. 1–36, 2014, ISSN: 15487660. DOI: `10.18637/jss.v061.i06`.

[40] L. Nováková and O. Štepanková, “Radviz and identification of clusters in multidimensional data,” in *2009 13th International Conference Information Visualisation*, IEEE, Barcelona, Spain: IEEE, 2009, pp. 104–109.

[41] J. Browniee, *Evaluate Peformance of Machine Learning Algorithms Using Resampling*, 2016. [Online]. Available: `https://machinelearningmastery.com/evaluate-performance-machine-learning-algorithms-python-using-resampling/` (visited on 10/13/2021).

Appendices

Listing 8 Bash script algorithm to run Sync Task Example code, collect JLM and perf data and store them

```
#!/bin/bash
javac -sourcepath ${BENCH_CLASS_DIR} ${BENCH_CLASS_DIR}/${BENCH_CLASS}.java

java -Xjit:perfTool -agentlib:jprof -classpath ${BENCH_CLASS_DIR} \
    ${BENCH_CLASS} ${THREADS} ${SLEEP_TIME} &> /dev/null &

# Capture pid of java program
PID_JAVA=`ps aux | grep 'agentlib:jprof' | grep -v grep | awk '{print $2}'`

if [[ "" !=  "$PID_JAVA" ]]
then
 # Record JLM data
 rtdriver -a 127.0.0.1 -c jlmstart 10 -c jlmdump 10 -c jlmstop &

 # Record perf data
 sleep 10
 perf record -p $PID_JAVA -g &

 sleep 10
 PID_PERF_REC=`ps aux | grep 'perf record' | grep -v grep | awk '{print $2}'`

 kill -SIGINT $PID_PERF_REC
else
  echo "java pid process not found!"
fi

# Kill the java program
kill -9 $PID_JAVA

# Convert raw perf.data to human-readable perf.log file
perf script -G -F comm,tid,ip,sym,dso | ./perf-hottest sym > perf.log

# Store the raw perf and JLM log
mv perf.log ./{DESIRED_PATH}/perf.log
mv jlm.xxx ./{DESIRED_PATH}/jlm.log
```

Listing 9 Bash script algorithm to run test multiple times varying thread number and sleep time

```
#!/bin/bash
for THREADS in {10,50,100,200,300,400,500,1000}
do
  TRIALS=200
  DELAY=10
  DELAY_TYPE='ns'
  for ((i = 0; i < $TRIALS; i++))
  do
    ./collect-log.sh /log_saving_path/ /class_path/ JavaMainClass \
        ${THREADS} ${DELAY} ${DELAY_TYPE}
    DELAY=$[$DELAY + 100]
  done
done
```