Lock Contention Performance Classification for Java Intrinsic Locks

Nahid Hasan Khan[1], Joseph Robertson[1], Ramiro Liscano[1(✉)], Akramul Azim[1], Vijay Sundaresan[2], and Yee-Kang Chang[2]

[1] Ontario Tech University, L1G 0C5 Oshawa, ON, Canada
{hasan.khan,joseph.robertson}@ontariotechu.net,
{ramiro.liscano,akramul.azim}@ontariotechu.ca
[2] IBM Canada, Toronto, ON, Canada
{vijaysun,yeekangc}@ca.ibm.com

Abstract. Improper management of locks and threads can lead to contention and can cause performance degradation and prevent software applications from further scaling. Nowadays, performance engineers use legacy tools and their experience to determine causes of lock contention. In this paper, 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 eventually support developers with less experience. The classifier is based on the premise that if lock contention exists it is reflected as either threads spending too much time inside the critical section and/or high frequency access requests to the locked resources. Our results show that a KMeans classifier is able to identify three classes of lock contention from run-time data where one of these classes is clearly caused by high hold times. The other two classes are more challenging to label but one appears to be caused by high frequency requests to the locked resource.

Keywords: Lock contention · Concurrency · Run-time faults · Classification · Software engineering

1 Introduction

Synchronization is essential in multi-threaded applications and introduces some level of thread contention when applied. When this contention is significant it results in performance degradation and is typically known as a lock contention fault or performance bottleneck due to contention.

It is difficult 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%

T. Dang and V. Stolz (Eds.): RV 2022, LNCS 13498, pp. 274–282, 2022.
https://doi.org/10.1007/978-3-031-17196-3_16

of all critical sections are changed at some point by the developers, both to fix correctness bugs and to enhance performance [7,14].

Lock contention bottlenecks have been investigated in the software community for a while but they are still difficult to detect [11,12] 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. The long term motivation of our work is to develop a recommendation system for software developers that encounter lock contention faults. We are proposing a combined approach that leverages a classification of the contention bottle neck based on those defined by Goetz [6] with the eventual goal of matching these contention types with patterns in the code. This paper focuses on preliminary work in the classification of lock contentions based on run-time performance metrics of the application.

In Goetz's book titled "Java Concurrency in Practice" he identifies the following 2 potential causes for contention faults:

- Type 1 - Threads spending too much time inside the critical sections.
- Type 2 - High frequency access requests by multiple threads to the locked resources.

The reason why these types are important to identify is that recommendations to alleviate the lock contention differs for types 1 and 2 [1]. In Type 1 the focus in on reducing the hold time on the lock while for Type 2 the solutions focus mostly on reducing the scope of the lock.

The paper is organized as follows. We introduce some related works in Sect. 2. The paper's main methodology is presented in Sect. 3. Section 4 describes the analysis of the clustering results. Some of the limitations of our approach and concluding remarks are presented in Sect. 5.

2 Related Works

Lock contention performance bottlenecks have been well investigated in the past few years with most works focusing on detecting and locating the root cause of the lock contention but very few papers have attempted to categorize the lock contention with the goal to suggest recommendations to alleviate the contention.

One of the few papers that tries to distinguish between type 1 and 2 lock contentions for categorizing and diagnosing synchronization performance faults is the work on SyncPerf by Mejbah ul Alam et al. [1]. In their work they reviewed several papers and categorized them in a quadrant based on the level of contention rate vs. lock acquisition frequency and concluded that there were no publications that addressed the detection of lock contention type 2 but focused primarily on the detection of locks of type 1.

Nathan R. Tallent et al. [13] details three approaches to gaining insight into performance losses due to lock contention. Their first two approaches use call

stack profiling and prove that this profiling does not yield insight into lock contention. The final approach used an associated lock contention attribute called thread spinning that helps provide insight into lock contention.

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 [3]. This paper stated that they failed to determine a correlation among the metrics extracted from the 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.

3 Methodology

Our approach uses run-time logs from several performance analyzers such as Linux *perf* [10], and JLM [9], then analyzes them utilizing a popular clustering technique (KMeans) to determine the existence of different types of contention faults. This analysis is a form of unsupervised classification and it was chosen over supervised classification because performance engineers do not label the types of lock contention faults.

A high-level workflow of our methodology is shown in Fig. 1. The approach can be divided into two main parts; the dataset creation and the feature engineering and classification. The principal processes in the dataset creation stage are the run-time performance data analysis and the data filtering. The run-time performance data analysis process consists of a compiled Java program that reflects a multi-threaded concurrent example integrated with a benchmark tool such as IBM Performance Inspector. The main process in the feature engineering and classification part of the workflow is the contention classification process that extracts from the performance data a set of clusters.

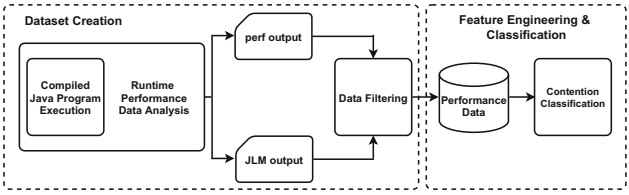


Fig. 1. High level workflow of our methodology where the steps are divided into two main parts, the Dataset creation and, the Feature engineering and classification.

3.1 Dataset Creation

Our methodology requires the generation of a dataset of lock contention performance metrics and currently no such dataset exist so the creation of such a

dataset is an important part of this work. This code is shown in Listing 1.1 and is executed in a controlled environment. The code is compiled before running, and necessary command-line arguments are provided prior to execution. The command-line arguments are the number of threads and the execution time of the critical section (emulated by putting the thread to sleep and shown in line 8 of the code.) The number of threads is set in the main program that executes this code. The code is executed multiple times to reduce the effects of outliers in the metrics, and we usually skip the first 10*s* of the execution to avoid the JVM's code optimization and warm-up period.

Listing 1.1. A Java example class "SyncTask" that is being used to emulate different types of faults in our controlled environment

```
1  class SyncTask {
2    public ArrayList<Integer> arrList;
3    public int sleep_t,
4
5    public synchronized void taskOne(Integer value){
6      try {
7        arrList.put(value);
8        Thread.sleep(sleep_t);
9      } catch(Exception e) {
10       e.printStackTrace();
11     }
12   }
13 }
```

There were two configurations that we used to generate the dataset: the first was 10, 100, 500, and 1000 threads with sleeptimes from 1 to 20000ns in 100ns increments, the second was 10, 50, 100, 200, 300, 400, 500, and 1000 threads with sleeptimes from 1 to 20000ns in 100ns increments.

The two configurations were created for convenience and to break up the total time it takes to create the dataset. One single run takes 40 to 45 s to complete and creates a single row in the dataset. As an example, configuration 1 takes about 9.4 h on average to generate the dataset. The output of the two configurations are combined to create the final dataset.

Once the Java code bench marking is completed, the *perf* and JLM performance analyzers are executed to collect the necessary run-time performance profile data. It is best to ensure that a high-performance machine with a bare-metal operating system is installed to execute the concurrent code and collect performance data. We installed these tools on a high-performance Linux machine with a 24 core processor (3800 MHz MHz) and 32 GB of RAM. For the Java environment, we use Openj9 JVM because it is compatible with JLM [4].

JLM provides quite a few metrics related to Java inflated monitors and a brief description of the most important of those metrics is helpful to the readers to understand how they relate to lock contention faults.

- GETS: Total number of successful acquires. GETS = NONREC + REC.
- TIER2: The number of inner loops to obtain locks.

- TIER3: The number of cycles in the outer layer to obtain the lock.
- %UTIL: Monitor hold time divided by total JLM recording time.
- AVER-HTM: Average amount of time the monitor was held.

When lock contention occurs, JLM lists the Java monitors under the "JLM Inflated Monitors" along with the specific values for each metric as defined above.

The *perf* tool is capable of capturing the symbols from system memory. These symbols are mainly method names, variables, or class names usually used in the OS itself, the kernel, or in the Java application. The reference of how the *perf* tool works can be found here [10]. With the help of a script, we managed to extract a human-readable log containing the following 3 columns of values; the sample count, the percentage of the sample count relative to the total sample counts, and the symbol name.

3.2 Feature Engineering and Classification

Feature engineering enhances the performance of the model and is essential for enhancing the results of the clustering algorithm [8]. In order to achieve this, the following initial data pre-processing is done prior to applying clustering techniques:

1. Merge the *perf* and JLM data files into one file and Python data frame.
2. Remove features that contain string values (e.g., the monitor name in JLM).
3. Remove features that contain a value of zero.
4. Scale the data. Scaling is applied to all the features utilizing the Python library `StandardScaler` from `sklearn.preprocessing`.
5. Remove correlated features from the dataset.

After performing the steps listed above the final dataset consisted then of the following twelve metrics: The metrics %MISS, GETS, NONREC, SLOW, TIER2, TIER3, REC, %UTIL and AVER_HTM are collected from JLM and the metrics "_raw_spin_lock", "ctx_sched_in" and "delay_mwaitx" are collected from the *perf* tool.

Determining Ideal Number of Clusters. The KMeans clustering algorithm requires that the number of clusters be specified prior to determining the clusters from a dataset. This value is known as the KMeans "k" value. In our research we determined the "ideal" number of clusters by calculating the Silhouette Coefficient [15] or silhouette score of the clusters.

Results of the silhouette scores, determined by leveraging the Python library `silhouette_score` from sci-kit learn `sklearn.metrics`, are shown in Fig. 2. The figure illustrates that a cluster number of 3 achieves the highest silhouette score against other cluster numbers so it can be considered as the "ideal" number of clusters. We have also validated that a cluster number of 3 is ideal using other methods such as the Elbow Method [5] and `NbClust` [2] package.

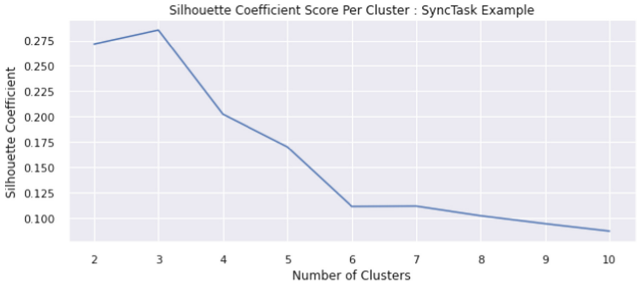


Fig. 2. The Silhouette Coefficient score for cluster number 3 is the highest. This indicates the optimal number of clusters is 3 that can be found within the dataset. Based on this verification, argument k = 3 is set to KMeans algorithm.

The KMeans algorithm of the Python library `KMeans` from `sklearn.cluster` was utilized to cluster the data set. The parameters "expected number of clusters" is set to three, "maximum number of iterations" is set to 600, and "minimum iterations" is set to 10. The extracted clusters are shown in Fig. 3.

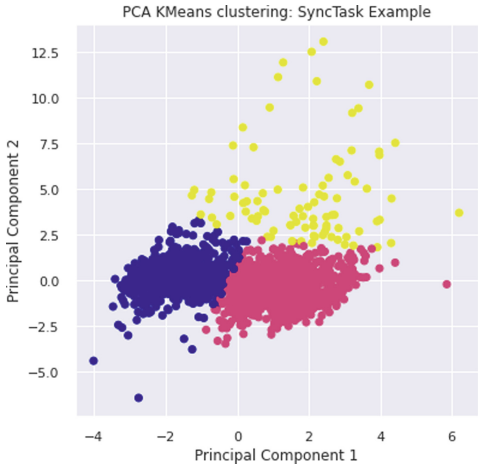


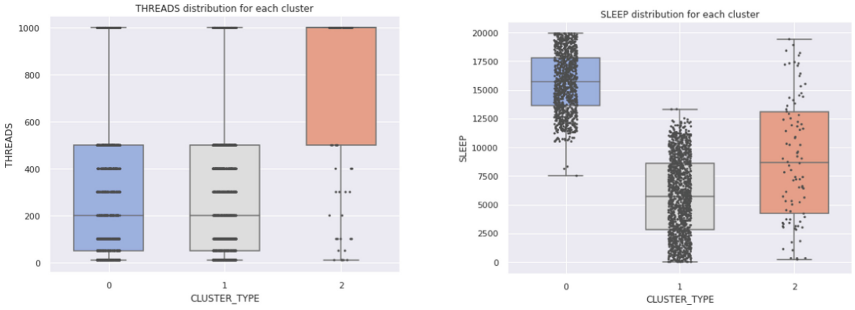
Fig. 3. Identified clusters from the lock-contention performance dataset using KMeans with a value of k = 3.

4 Cluster Analysis

In our data set 3 clusters were discovered and are numbered 0 to 2 but it is not clear if any of these clusters represent the lock contention types defined by Goetz [6]. For this simple example on can determine this by considering the distribution

of the sleep time and number of threads in the clusters. Hypothetically, fault type 1 (fault due to high hold-time) depends on the provided variable parameter sleep times during the execution of the concurrent code. On the other hand, fault type 2 (fault due to high frequent access requests by the threads) should depend on a high number of threads and low sleep times.

Hence we plot the "THREADS" and "SLEEP" distribution against the clusters utilizing a box plot and observe the results. The two graphs are shown in Fig. 4a and Fig. 4b. CLUSTER_TYPE = 0 possesses a higher range of sleep times, indicating that more likely it represents fault type 1. On the other hand for the other 2 clusters it is challenging to label one of these as fault type 2. The figures illustrate that CLUSTER_TYPE = 2 contains a high number of threads compared to the other 2 clusters but the sleep time is slightly higher than CLUSTER_TYPE = 1. With this knowledge we believe that CLUSTER_TYPE = 2 can be labelled as fault type 2 while CLUSTER_TYPE = 1 is a form of low contention.



(a) (b)

Fig. 4. Box plot visualization of the Threads (a) and Sleep (b) parameters related to the clusters to help label the clusters.

In order to get a better sense of the semantic meaning of the clusters it helps to observe the dominant features in the clusters. Figures 5 show 4 box plots of the JLM metrics AVG_HTM, GETS, TIER2, and TIER3 relative to the 3 clusters. The AVG_HTM and GETS metrics are typically negatively correlated and one can observe this in CLUSTER_TYPES 0 and 1. This is not that obvious for CLUSTER_TYPE 2 as statistically these 2 values are indistinguishable.

The AVER_HTM value is a significant feature and from Fig. 5a one can see that the AVER_HTM value is higher for fault type 1 (CLUSTER_TYPE 0) than the other clusters. Moreover, the figure also illustrates that CLUSTER_TYPE 1 has a lower value than that of CLUSTER_TYPE 2 but the difference is not significant. This could imply that CLUSTER_TYPE 1 represents fault type 2 better than CLUSTER_TYPE 2 though when one looks at the threads distribution it is on par to those for CLUSTER_TYPE 0 and lower than those for CLUSTER_TYPE 2.

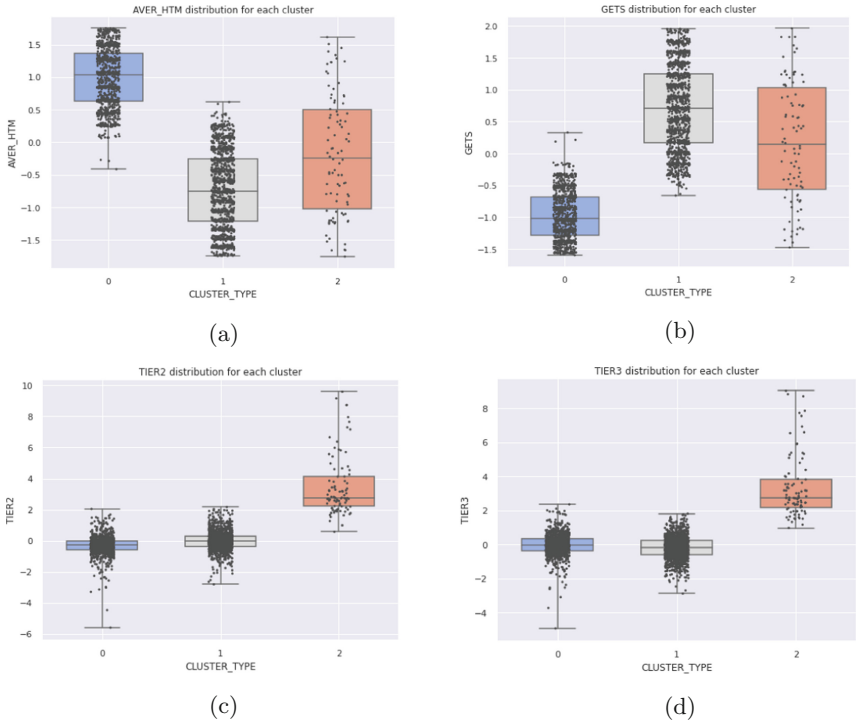


Fig. 5. Box plot visualization of the dominant features for each of the clusters. The AVER_HTM feature (a), the GETS feature (b), the TIER2 feature (c), and the TIER 3 feature (d).

Our expectation for the features related to spin counts (TIER2 and TIER3) are that they should experience high numbers for high lock request rates (fault type 2). The results in Fig. 5c and Fig. 5d show that CLUSTER_TYPE 2 has higher spin counts values than the other 2 clusters and presents more evidence that CLUSTER_TYPE 2 represents fault type 2.

After this analysis we conclude the following: (1) "Contention Fault 1" has high hold times, (2) "Contention Fault 2" has low in hold times and high spinning counts, (3) CLUSTER_TYPE 1 is more likely a form of "Low Contention", and (4) CLUSTER_TYPE 1 is more likely a form of "Low Contention" since it contains low spinning counts as well as low hold times.

5 Conclusions

In this paper, we demonstrate that Java intrinsic locks could be classified into the two types and we used a unsupervised KMeans classifier to investigate the types using run-time performance metrics. We find lock-contention faults do appear to classify into three distinct clusters rather than two based on run-time performance metrics where fault type 1 can be clearly identified due to the high hold times while fault type 2 is more challenging to identify as there is no cluster

with a clear high number of threads and low hold time. The three clusters are differentiated by the hold times and spinning counts and this knowledge can be used to train a decision tree to help identify lock contention types for other Java applications.

References

1. Alam, M.M.U., Liu, T., Zeng, G., Muzahid, A.: Syncperf: categorizing, detecting, and diagnosing synchronization performance bugs. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 298–313 (2017)
2. Charrad, M., Ghazzali, N., Boiteau, V., Niknafs, A.: Nbclust: an R package for determining the relevant number of clusters in a data set. J. Stat. Softw. **61**(6), 1–36 (2014). https://doi.org/10.18637/jss.v061.i06
3. David, F., Thomas, G., Lawall, J., Muller, G.: Continuously measuring critical section pressure with the Free-Lunch profiler. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, vol. 49(10), pp. 291–307 (2014). https://doi.org/10.1145/2660193.2660210
4. Eclipse Foundation, I.: OpenJ9 (2017). https://www.eclipse.org/openj9/
5. Franklin, J.S.: Elbow method of K-means clustering using Python - Analytics Vidhya - Medium (2019). https://medium.com/analytics-vidhya/elbow-method-of-k-means-clustering-algorithm-a0c916adc540
6. Göetz, B., Professional, A.W.: Java concurrency in practice. Building **39**(11), 384 (2006)
7. Gu, R., Jin, G., Song, L., Zhu, L., Lu, S.: What change history tells us about thread synchronization. In: 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). https://doi.org/10.1145/2786805.2786815
8. Hale, J.: Scale, Standardize, or Normalize with Scikit-Learn (2019). https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02
9. IBM: Java Lock Monitor (1999). http://perfinsp.sourceforge.net/examples.html#jlm
10. Kernel.org: Linux kernel profiling with perf (2015). https://perf.wiki.kernel.org/index.php/Tutorial
11. Kohler, M.: A Simple Way to Analyze Thread Contention Problems in Java (2006). https://blogs.sap.com/2006/10/18/a-simple-way-to-analyze-thread-contention-problems-in-java/
12. Salnikov-Tarnovski, N.: Improving Lock Performance (2015). https://dzone.com/articles/improving-lock-performance
13. Tallent, N.R., Mellor-Crummey, J.M., Porterfield, A.: Analyzing lock contention in multithreaded applications. ACM SIGPLAN Notices **45**(5), 269–279 (2010). https://doi.org/10.1145/1837853.1693489
14. Yu, T., Pradel, M.: Pinpointing and repairing performance bottlenecks in concurrent programs. Empir. Softw. Eng. **23**(5), 3034–3071 (2017). https://doi.org/10.1007/s10664-017-9578-1
15. Zhou, H.B., Gao, J.T.: Automatic method for determining cluster number based on silhouette coefficient. In: Advanced Materials Research. Advanced Materials Research, vol. 951, pp. 227–230. Trans Tech Publications Ltd, Switzerland (2014). https://doi.org/10.4028/www.scientific.net/AMR.951.227