Identification of Java lock contention anti-patterns based on runtime performance data

Aritra Ahmed
aritra.ahmed@ontariotechu.net
Ontario Tech University
Oshawa, Ontario , Canada

Ramiro Liscano
rliscano@ieee.org
Ontario Tech University
Oshawa, Ontario , Canada

Akramul Azim
Akramul.Azim@ontariotechu.ca
Ontario Tech University
Oshawa, Ontario , Canada

Yee-Kang Chang
yeekangc@ca.ibm.com
IBM Canada
Toronto, Ontario, Canada

Vijay Sundaresan
vijaysun@ca.ibm.com
IBM Canada
Toronto, Ontario, Canada

ABSTRACT

Locks play a crucial role in multi-threaded applications, offering an effective solution for synchronizing shared resources. Yet, mishandling locks and threads can result in contention, leading to performance deterioration and compromising the scalability of software applications. In this study, several machine learning models were evaluated on how well they could detect the Java lock contention anti-pattern that caused the lock contention fault based on run time performance data. We trained the machine learning models with performance data generated from the execution of eight Java lock contention anti-patterns and tested the prediction of the models against 30% of the training data as well as performance data from six applications in the Dacappo benchmark that exhibit lock contention. Our results show that we can accurately identify the lock contention anti-pattern based on runtime performance data with an accuracy close to 90%.

KEYWORDS

Lock Contention Faults, Lock Contention Anti-patterns, Supervised Machine Learning

ACM Reference Format:
Aritra Ahmed, Ramiro Liscano, Akramul Azim, Yee-Kang Chang, and Vijay Sundaresan. 2024. Identification of Java lock contention anti-patterns based on runtime performance data. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3644032.3644466

1 INTRODUCTION

The Java programming language incorporates multi-threading through an intrinsic lock or monitor lock, managed by an Application Programming Interface (API) [15]. This lock system, inherent to every Java object, regulates synchronized operations and access, with the Java Virtual Machine internally implementing the locking mechanism for thread synchronization [10]. Despite the necessity of synchronization in multi-threaded applications, contention arises when threads vie for shared resources, leading to performance degradation or a lock contention fault. This contention can impede application performance, highlighting the need for effective management in concurrent programming.

This work focuses on the effectiveness of machine learning models to identify the lock contention Java code anti-pattern based on the runtime performance data captured from tools like Java Lock Monitor (JLM) [7] and *perf* [9].

Unlike many other approaches in the literature that have investigated lock contention faults, our focus is on the identification of the lock contention anti-pattern with the goal of presenting software developers with alternative code change recommendations.

In this study, the machine learning models are trained using performance data generated from 8 Java anti-patterns related to lock contention. These examples include coarse-grained locking, unified locking, loop inside critical section, loop outside critical section, nested locking, overly split locks, improper access, and using the same lock in multiple methods. Each anti-pattern is described with associated recommendations, with a focus on issues relating to Java intrinsic locks.

The paper is organized as follows. We introduce some related works in section 2. The paper's main methodology is presented in section 3. The experimental results demonstrating the effectiveness of the different machine learning models models on predicting the lock contention anti-patterns is presented in section 4. We discuss the future works and conclude the paper in section 5.

2 RELATED WORKS

In our background study we did a comprehensive research and reviewed previous works that have focused mainly on lock contention detection and localization using dynamic analysis approaches.

The work by Mejbah ul Alam et al. [1] and their *SyncPerf* tool is one of the closest related to our approach in that they classify lock contention into 6 different lock types in order suggest code change recommendations to alleviate the lock contention though they do not describe how their detection tool determines which class a lock contention belongs to.

Chen Zhang et al. [22] implemented a static synchronization performance bug detection tool that identified critical sections, different loops, and pruning components. Using three real-world distributed cloud systems, they collected 26 performance bugs and successfully detected them with their tool. Although the area of focus is different, the paper's analysis aligns with our approach.

The main focus of the other papers that we have reviewed included dynamic analysis approaches. These works included tracing locking events in JVM, emphasizing runtime logs and metrics to analyze contention faults and address bottlenecks, measuring critical section pressure for fault localization [6, 21, 22]. Most of these works focus on locating the cause of the fault but fail to offer any recommendation to mitigate the lock contention.

There were previous work relating to lock contention to anti-patterns/bug patterns/code smell and these studies mainly focuses on static analysis which implies to studying the source code of the applications to detect the faults and providing fixing strategies [4, 5, 11, 12].

3 METHODOLOGY

To recap our aim is to build robust machine learning models that can predict lock contention code anti-patterns from run time performance data. Once the anti-pattern is identified the anti-pattern recommendations can be presented to a developer in order to mitigate the lock contention.

Figure 1 depicts a high-level methodology diagram for the steps involved to train the machine learning algorithms. It is composed of 2 significant phases consisting of the dataset creation followed by the a feature engineering and classification phase.

The first process in the dataset creation involves the compilation and execution of a set of Java anti-pattern lock contention example codes and leveraging the performance metric tools JLM and *perf* to capture runtime performance data.

Next the dataset is labeled so that each performance entry is assigned a number corresponding to a specific lock contention anti-pattern.

Finally, particular features from the data set are selected in order to train a machine learning model.

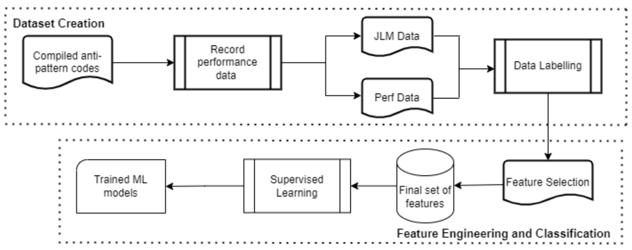

Figure 1: High-level approach to train the ML models

3.1 Dataset Creation

Code execution is the first phase of the dataset creation and in order to do this eight Java anti-pattern code examples, summarized in table 1, are compiled and executed with arguments that define the number of threads created of the example anti-pattern as well the execution time spent within a critical section of the code example (emulated by putting the thread to sleep). The Java code execution triggers *perf* and JLM to collect runtime performance data as the lock contention example codes are executed.

Table 1 lists the lock contention anti-patterns that were used for the creation of the performance training dataset along with their lock contention anti-pattern identifier. All of the examples represent typical code patterns that leverage the Java implicit locks. Due to limitations in the page limits of the paper detail code examples of the lock contention anti-patterns have been left out and their titles and a brief descriptions are shown in the table. The anti-patterns are a revised set inspired by Robertson's thesis on Java Lock Contention Anti-patterns [16].

The following configuration was considered to generate the dataset:

- **Configuration:**
  - Threads: [8, 16, 32, 64, 128, 256, 1024]
  - Sleep: From $10ns$ up to $20,000ns$ in $100ns$ increments, with 200 runs
  - Total Data Points: 1400

The thread counts are chosen to diversify the dataset.

For concurrent code execution and data collection, we use an isolated environment on a high-performance Linux machine with a 24-core processor (3800 MHz) and 32 GB RAM, using Openj9 JVM. The JLM log focuses on contention statistics related to Java monitors.

A single run takes 40-45 seconds, creating a row. With 1400 rows per sample, a sample takes about 17.5 hours to be created resulting eventually in a 11,200-row dataset.

The example codes are executed multiple times to reduce the effects of outliers in the metrics, and we skip the first 10 seconds of the execution to avoid the JVM's code optimization and warm-up period.

3.2 Performance Metrics

JLM provides several metrics related to Java inflated monitors and we have used the following key metrics in our research:

- GETS: Total number of successful lock acquires.
- SPIN_COUNT: A summation of the total number of loop iterations to obtain a lock.
- AVER-HTM : Average amount of time the monitor was held.
- %UTIL : The total time the monitor was held divided by total JLM data acquisition time.

The *perf* tool captures memory symbols from system memory, like method names, variables, or class names in the OS, kernel, or Java application. Raw *perf* data, though unreadable, is processed into a human-readable log with sample count, percentage relative to total samples, and symbol name. In our work we capture the "_raw_spin_lock," "ctx_sched_in," and "delay_mwaitx" from *perf* as these are the ones closely related to lock contention. All metrics are numerical.

Feature engineering enhances the performance of the model and is essential for enhancing the results of the algorithms. 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.

No.	Anti-pattern Name	Description	Recommendation
1	LOC	Loop outside critical section leading to lock contention	Assessing the loop for potential optimizations, such as parallelizing independent operations and implementing batch processing for multiple items
2	Unified locking	Same lock used in independent variables	Lock Striping, using separate locks for independent variables
3	Nested Locking	Multiple locks inside a method, potential of circular dependencies among threads	Lock Ordering or re-entrant locks should be employed which means that all threads must adhere to a standardized sequence of lock acquisition for the shared resources they need
4	Coarse-grained	Locks entire methods instead of objects	Synchronization on objects rather than methods
5	LIC	Loop inside critical section leading to lock contention	Parallelizing the loop, breaking down the loop into smaller tasks which can be executed in parallel by distinct threads or processes
6	Same lock	Same lock is shared among different methods in a class	Change from object-based synchronization to atomic type
7	Overly Split	Lock granularity way too finer than necessary	Merging the split locks into more cohesive chunks
8	Improper Access	Accidental access of a variable without appropriate protection	Different approaches based on the error

Table 1: Definition of anti-patterns, recommendations for the anti-patterns and their associated numbers

(2) Remove features that contain string values (e.g., the monitor name in JLM). The clustering algorithms require data to be numerical and tabular.
(3) Remove features that contain a value of zero.

3.3 The Machine Learning Models

We trained the following machine learning models, **Random Forest** [18], **Logistic Regression** [13], **Multinomial Naive Bayes** [8], **Support Vector Machine (SVM)** [2], **K-Nearest Neighbor (KNN)** [19] and **XGBoost** [14] with specific configurations and experimented with a few ranges of hyper-parameters before training the models. To optimize machine learning, we tuned hyper-parameters using K-fold cross-validation and grid search with the GridSearchCV [17] class. Using nested cross-validation enhanced model robustness by iteratively fine-tuning hyper-parameters within the broader K-fold procedure. The final model, refitted with the best hyper-parameters (HP), ensures improved performance on the entire training dataset.

The table 2 shows the values we used after the mentioned experiments ensuring best performance from the models.

ML Models	Input Parameters	Best HP value
Random Forest	n_estimators	154
	max_depth	20
	min_samples_split	7
	max_features	3
Logistic Regression	penalty	I2
	C	1.0
	max_iter	1000
Multi NB	alpha	1.0
Support Vector Machine	kernel	linear
	C	1.0
KNN	n_neighbors	7
XG Boost	n_estimators	170
	learning_rate	0.1
	max_depth	5
	subsample	0.8

Table 2: The input parameter grids for the ML models and the best hyper-parameters obtained using the GridSearchCV algorithm

4 EXPERIMENTAL RESULTS

4.1 Evaluation of the models using the training dataset

The performance data from anti-patterns was utilized to train machine learning models, employing Python libraries for the process. The dataset was taken into a pandas dataframe, excluding the 'Pattern No' column to form the feature matrix 'X'. The 'Pattern No' column was selected to create the label array 'y'. The dataset was split into 70-30% training ('X_train', 'y_train') and testing ('X_test', 'y_test') sets. Features were standardized using StandardScaler to have a mean of 0 and standard deviation of 1, ensuring consistency across algorithms reliant on distance metrics. 'X_train' was standardized using the mean and standard deviation from the training set, while 'X_test' was standardized using the same parameters computed from the training set.

The performance of the described machine learning algorithms were evaluated against the testing dataset. The bar plot shows the accuracy of all the algorithms. All the algorithms except multinomial Naive Bayes has an high accuracy over 90%.

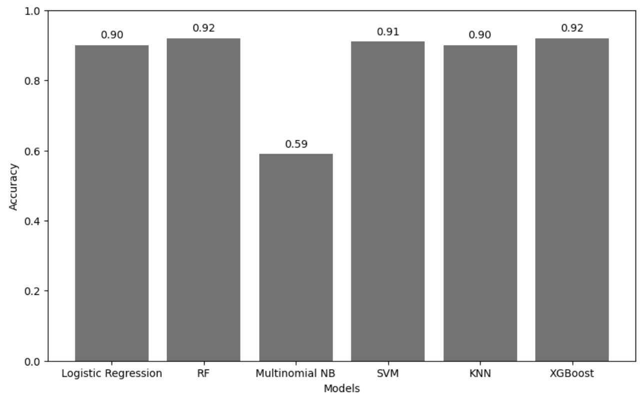

Figure 2: Accuracy plot of all the 6 machine learning algorithms

4.2 Evaluation of the models using the DaCapo Benchmark Suite

In this part of the project we have worked with the DaCapo Benchmark Suite[3, 20]. It consists of a set of open-source Java programs and benchmarks that are widely used to assess the performance of Java applications and the underlying JVMs.

The benchmarks *avrora*, *h2*, *jython*, *lusearch*, *pmd*, and *xalan* were executed and runtime performance metrics were captured using JLM and perf. These particular benchmarks were chosen because they exhibited lock contention behavior in a multi-threaded environment and it was possible to label the lock contention anti-pattern that caused the lock contention.

To conduct meaningful testing using the DaCapo benchmarks, we required the benchmarks to be labeled. This was performed manually by the principal author of the paper by looking at the code of a critical region and deciding which anti-pattern it closely resembled.

To collect performance data from these benchmarks, we ensured consistency in the parameters used to generate the dataset, except for the ability to configure the number of threads. We tested each benchmark with 7 different thread variations, mirroring the training setup, and repeated each thread configuration 5 times, resulting in 35 data points for each benchmark.

The Benchmarks must run for at least 20s to capture JLM and *perf* metrics; setting the *maximum iteration* to 200 ensuring sufficient time for data collection.

Table 3 show the predicted anti-patterns for the DaCapo benchmarks associated with each of the machine learning models. Within the table, the benchmarks were assigned an anti-pattern number, indicative of their labeling.

The weighted average and accuracy of the models is shown in figure 3. The accuracy metrics is calculated from the ratio of correctly predicted instances to the total instances in the dataset. In scenarios with uneven class distributions, the weighted average metric offer a better evaluation which makes it more important than accuracy. While labeling the benchmarks we had more instances in one class of antipattern than the others. This approach calculates the average of metrics for each class, weighted by the number of true instances, providing a comprehensive assessment of classification performance.

Benchmarks		Predictions				
	Actual Label	LR	RF	SVM	KNN	XGB
avrora	5	5	4	5	5	4
lusearch	1	4	1	4	1	1
pmd	4	4	4	4	4	4
h2	4	4	4	4	3	4
jython	4	4	4	4	4	4
xalan	4	4	4	4	4	4

Table 3: The predicted labels for the benchmarks from the machine learning algorithms

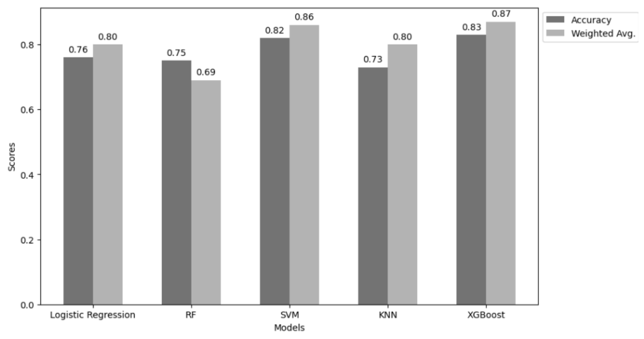

Figure 3: Grouped bar plot showcasing the accuracy and weighted average values of the algorithms from the benchmark testing set.

5 CONCLUSIONS

In this research, our goal was to explore the ability to identify Java lock contention anti-patterns from runtime performance traces by using machine learning models. We trained six machine learning algorithms for multi-class classification using performance data generated from the execution of example code of Java lock contention anti-patterns.

The results show that all machine learning models, except the Multinomial Naive Bayes, achieved over 90% accuracy when tested against 30% of the dataset generated from the JAva lock contention anti-patterns. When the models were tested against the DaCapo benchmarks the SVM and XGBoost models achieved a prediction in the 87% range.

In the future, we aim to focus on the recommendations that the lock contention anti-patterns offer and the generation of refactored code from the course code.

ACKNOWLEDGEMENTS

This research was supported in part by IBM Center for Advanced Studies (CAS) and Natural Sciences and Engineering Research Council of Canada (NSERC) grants.

REFERENCES

[1] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 298–313. https://doi.org/10.1145/3064176.3064186

[2] baeldung. 2018. Multiclass Classification Using Support Vector Machines. https://www.baeldung.com/cs/ svm-multiclass-classification.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. 2003. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience* 15, 3-5 (2003), 485–499. https://doi.org/10.1002/cpe.654 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.654

[5] H. H. Hallal, E. Alikacem, W. P. Tunney, S. Boroday, and A. Petrenko. 2004. Antipattern-based Detection of Deficiencies in Java Multithreaded Software. In *Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings* (Braunschweig, Germany, 08-09 September 2004). IEEE. https://doi.org/10.1109/QSIC.2004.1357968

[6] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. 2016. 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* 2016-March (2016), 263–274. https://doi.org/10.1145/2851553.2851559

[7] IBM. 1999. Java Lock Monitor. http://perfinsp.sourceforge.net/examples.html#jlm

[8] Idil Ismiguzel. 2017. Naive Bayes Algorithm for Classification. https://towardsdatascience.com/ naive-bayes-algorithm-for-classification-bc5e98bff4d7.

[9] Kernel.org. 2015. Linux kernel profiling with perf. , 39 pages. https://perf.wiki.kernel.org/index.php/Tutorial

[10] Thin Locks. 1998. Thin Locks: Featherweight Synchronization for Java. In *Proceeding of the ACM COnference on Programming Language Design and Implementation, SIGPLAN*, Vol. 33. Association for Computing Machinery, New York, NY, USA, 258–268.

[11] Md Abdullah Mamun, Aklima Khanam, Håkan Grahn, and Robert Feldt. 2010. Comparing Four Static Analysis Tools for Java Concurrency Bugs. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*.

[12] Naouel Moha, Gueheneuc Yann-Gael, Duchien Laurence, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[13] R Nasrin. 2020. Multiclass Classification Using Logistic Regression from Scratch in Python: Step by Step Guide. https://towardsdatascience.com/multiclass-classification-algorithm -from-scratch-with-a-project-in-python step-by-step-guide-485a83c79992.

[14] Ernest NG. 2020. XGBoost for Multi-class Classification. https://towardsdatascience.com/xgboost-for- multi-class-classification-799d96bcd368.

[15] Oracle. 2015. Intrinsic Locks and Synchronization (The Java™ Tutorials > Essential Classes > Concurrency). https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

[16] Joseph Robertson. 2023. Java Lock Contention Antipatterns and Their Detection within Java Code. https://ir.library.ontariotechu.ca/bitstream/handle/10155/1619/Robertson_Joseph.pdf?sequence=1

[17] scikit learn. 2020. Grid Search Cross Validation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[18] A Shafi. 2021. Random Forest Classification with Scikit-Learn. https://www.datacamp.com/tutorial/ random-forests-classifier-python.

[19] Vatsal Sheth. 2018. MultiClass Classification Using K-Nearest Neighbours. https://towardsdatascience.com/multiclass-classification-using-k-nearest-neighbours-ca5281a9ef76.

[20] sourceforge. 2019. DaCapo Benchmarks Home Page. https://dacapobench.sourceforge.net/.

[21] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. *ACM SIGPLAN Notices* 45, 5 (2010), 269–279. https://doi.org/10.1145/1837853.1693489

[22] Chen Zhang, Jiaxin Li, Dongsheng Li, and Xicheng Lu. 2019. Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems. *IEEE Access* 7 (2019), 99123–99135. https://doi.org/10.1109/ACCESS.2019.2923956