Towards Detecting Software Performance Anti-patterns using Classification Techniques

Manjula Peiris
James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN, USA
{tmpeiris, hillj}@cs.iupui.edu

ABSTRACT

This paper presents a non-intrusive machine learning approach called *Non-intrusive Performance Anti-pattern Detecter (NiPAD)* for identifying and classifying software performance anti-patterns. NiPAD uses only system performance metrics—as opposed to analyzing application level performance metrics or source code and the design of a software application to identify and classify software performance anti-patterns within an application. The results of applying NiPAD to an example application show that NiPAD is able to predict the One Lane Bridge software performance anti-pattern within a software application with 0.94 accuracy.

Keywords

software performance anti-patterns, classification, machine learning, dynamic software analysis

1. INTRODUCTION

Software performance anti-patterns [1] are common designs that have negative impact on software performance. Smith et al. [1] has identified several commonly occurring software performance anti-patterns. For example, *One Lane Bridge* occurs when a program is designed such that one, or only a few, processes may execute concurrently. Likewise, *Excessive Dynamic Allocations* occurs when a software application is frequently dynamically allocates and deallocates memory.

Detecting and resolving software performance anti-patterns is critical to producing high quality software systems. This is because failure to remove a software performance anti-pattern will cause system developers to seek alternative methods for resolving the corresponding problem, such as adding more hardware to coup with slow response times or rebooting the system to address memory problems.

Even though software performance anti-patterns are well-documented, the most prominent method for detecting and resolving them is (manual) source code analysis [2]. Unfortunately, source code analysis requires expert domain knowledge. Likewise, the source code may not be available to support such analysis. Finally, software performance anti-patterns occur in executing software. It is therefore *hard* to identify, evaluate, and resolve software performance anti-patterns using source code alone.

Another approach for detecting software performance anti-patterns is to use *dynamic software analysis*, which is the process of analyzing a software system by executing it. Although dynamic software analysis can address the limitations of source code analysis, existing dynamic software analysis approaches are either architecture/deployment dependent [3] or requirement specification dependent [4]. Other dynamic software analysis approaches use software instrumentation to analyze performance issues of the system [5–8]. These approaches, however, do not focus on identification and classification of software performance anti-patterns.

To address the limitations of existing dynamic software analysis techniques discussed above, we propose a non-intrusive approach for detecting software performance anti-patterns using only system performance metrics, such as CPU utilization. We call the approach *Non-intrusive Performance Anti-pattern Detecter (NiPAD)*.

NiPAD defines the software performance anti-pattern detection problem as a binary classification problem of system performance metrics. Given two sets of performance metric data where one is generated from a program that has a software performance anti-pattern and other from a program that does not have the software performance anti-pattern, NiPAD should be able to train a binary classification technique for that data set. The trained model then is used to predict future metrics with unknown labels (*i.e.*, is the metric is generated from a software application that has an software performance anti-pattern). Lastly, preliminary results from applying NiPAD to Apache web server shows that NiPAD can detect the One Lane Bridge with 0.94 accuracy using Support Vector Machines (SVM) [9].

Paper organization. The remainder of this paper is organized as follows: Section 2 discusses the intuition and functionality of NiPAD; Section 3 compares results of classification techniques used in NiPAD; Section 4 discusses related works; and Section 5 provides concluding remarks.

2. OVERVIEW OF NIPAD

The main intuition behind NiPAD is given two performance metric data sets from a software applications where one data set does not have a software performance anti-pattern and the other data set has a software performance anti-pattern, we should be able to find a discriminant function to separate the two data sets. This two data sets can be from two different software applications, or from the same software application with two different configurations.

Based on this intuition, NiPAD's approach can be divided into two phases. In the first phase, a classifier (see Table 1) is trained using a data set with a known class label (*i.e.*, has or does not have software performance anti-pattern). In the second phase, the classifier is used to predict the class label of new performance metrics. Finally, this process is repeated for different software performance anti-patterns.

2.1 Performance Metrics Classification Inputs

NiPAD uses system-level metrics, which show how the application is impacting the system where it is executing, as input to its classifiers. Examples of system-level metrics include CPU time, memory usage, and network usage. NiPAD uses system-level metrics because it is less intrusive to collect when compared to application-level metrics, which requires instrumenting the application.

Before NiPAD can definitive use system-level metrics to classify software

Table 1: Different classifiers implemented in NiPAD.

Classifier	**Description**
Logistic Regression	Calculating the probability of a class labels.
Fisher's Linear Discriminant	Maximizing mean difference between classes and minimizing within class variance
Naive Bayes Classifier	Maximizing the class conditional probabilities using Bayes theorem
Support Vector Machines (Linear)	Maximum margin classifier methods
Support Vector Machines (RBF)	Maximum margin classifier methods, and using a kernel function

performance anti-patterns, however, we first have to determine if there is enough variation in the data. This is important because it helps to separate two data sets with a higher accuracy using a discriminant function.

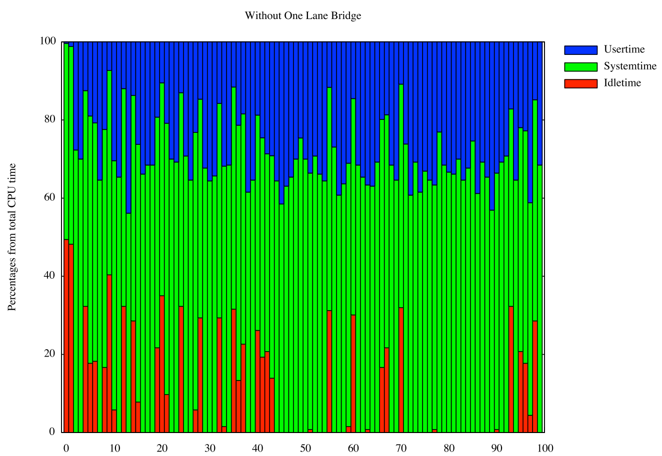

Figure 1: CPU times without the One Lane Bridge software performance anti-pattern.

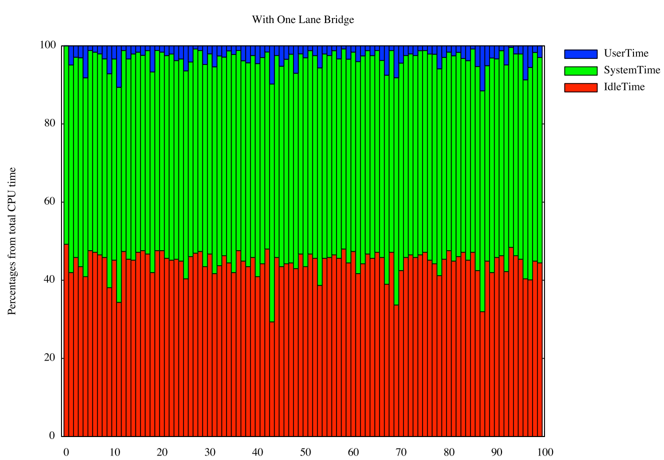

Figure 2: CPU times with the One Lane Bridge software performance anti-pattern.

One Lane Bridge Data. Figure 1 and Figure 2 illustrates the percentage of CPU time (*i.e.*, user time, system time, and idle time) for different *time epochs* (*i.e.*, when CPU time is sampled) when the One Lane Bridge is *not* present versus when it is present in the same software application. As shown in Figure 1, CPU idle time is low when the software application does not have the One Lane Bridge. On the other hand, CPU idle time is more—if not dominate—when the software application has the One Lane Bridge. Lastly, the variation in CPU time is attributed to the software application not utilizing the parallel capabilities (*e.g.*, all available CPU cores) of the system when One Lane Bridge is present.

This simple observation gives us confidence that NiPAD should be able to use system-level metrics to correctly classify the (non-)existence of software performance anti-patterns, such as the One Lane Bridge.

3. PRELIMINARY RESULTS OF NIPAD

We used *Apache Web Server* (`httpd.apache.org`) to evaluate NiPAD, and what classifiers (see Table 1) perform best. We used Apache Web Server because we can emulate different software anti-patterns by changing the server's configuration. We used *Apache Benchmark (AB)* (`httpd.apache.org/docs/2.2/programs/ab.html`) to generate workload. Finally, we collected system-level performance metrics while the server processed generated workload. More details on experiment setup and execution are explained within the results below.

3.1 Experimental Prediction of One Lane Bridge

The Apache web server uses thread pools to concurrently handle requests from many clients. When the server does not have enough threads (or resources) to handle requests, the requests are queued and clients typically block until its request is handled. Because the server has pending requests, it is considered to be not optimally utilizing its available resources. This is just one example of the One Lane Bridge [1].

To produce a data set the represents the example above, we configured Apache Web Server to have 150 threads. When then used AB to generate the workload of 300 concurrent clients that sent 1 million requests collectively. We labeled this data set *class 0*, or the negative class. To produce a data set without the software performance anti-pattern, we configured Apache Web Server to have 300 threads. We then used AB to generate the workload of 300 concurrent clients that sent 1 million requests collectively. We labeled this data set *class 1*, or the positive class. Lastly, we collected CPU times at 1 second time epochs while creating either data set. One-third of the data set was used for training and the remaining data was used for testing.

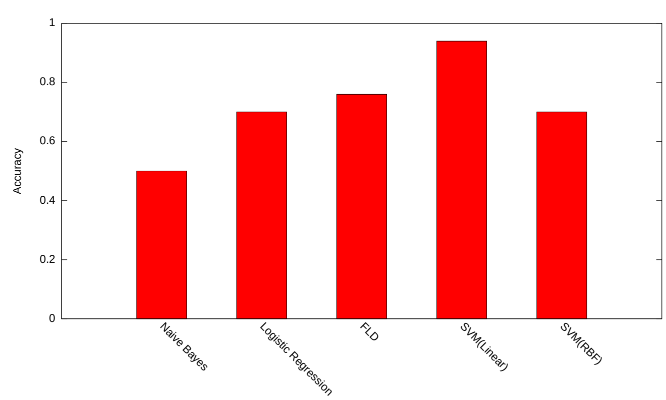

Figure 3: Comparison of accuracy in identifying the One Lane Bridge using NiPAD.

Figure 3 shows the accuracy comparison for identifying the One Lane Bridge. As shown this figure, Naive Bayes with Gaussian distribution using sample mean and variance of observed data as input parameters for class conditional probabilities performed the worse. Likewise, the SVM classifier with a linear kernel performed the best. This result is not surprising because Naive Bayes technique is the baseline technique used to compare different classification techniques. SVM surpass the other classification techniques because of its maximum margin concept, which gives better results in our experiments. SVM with RBF kernel could not produce better results than linear kernel because of training data overfitting.

Cost analysis of misclassification. Table 2 shows the calculated sensitivity (*i.e.*, ratio of identified actual positive labels to all actual positive labels); specificity (*i.e.*, ratio of identified actual true negative labels to all actual negative labels); and accuracy for each classifier in NiPAD. As shown in this table, sensitivity for the prediction is high. This implies that the classification techniques accurately predict the positive class (*i.e.*, the software application does *not* have the software performance anti-pattern). Compared to the sensitivity, specificity is low and misclassifications occur mostly for the negative class.

Classifier	Sensitivity	Specificity	Accuracy
Logistic	0.95	0.62	0.76
FLD	0.95	0.66	0.7
Naive Bayes	0.92	0.28	0.5
SVM (Linear)	0.98	0.92	0.94
SVM (RBF)	0.96	0.7	0.75

Table 2: Classifier performance analysis using different measures.

When a classifier predicts the existence of a software performance anti-pattern, then it is indication that the software's design needs refactoring. Unfortunately, refactoring is a costly, time-consuming task. Because NiPAD predicts situations where the software anti-patterns does not exist, it can eliminate unnecessary refactoring efforts.

3.2 Experimental Prediction of One Lane Bridge with Noise

The experiment in Section 3.1 was conducted in an isolated environment, *i.e.*, there were no other applications competing for resources). A more realistic situation is executing multiple applications on the same machine and competing for the same resources as the application undergoing performance analysis. We call the extra applications *noise*.

We created this scenario by periodically executing a second application on the same machine. The second application was programmed to utilize up to 30% of the CPU. While the second application was executing, we executed Apache Web Server using the same method as discussed in Section 3.1. Finally, we collected collected performance data using the same methods discussed before.

Figure 4 shows NiPADS's results for the data set with noise. As shown in this figure, the accuracies of each classifier decreased. The linear SVM classifier, however, still produced the highest accuracy of 0.74. We attribute this to less overfitting of training data to the model.

3.3 Experimental Prediction of Excessive Dynamic Allocations

The Excessive Dynamic Allocation occurs when a software application is frequently managing memory using system-level calls. One solution for this software performance anti-pattern is to manage memory using a preallocated memory pool [10]. We can recreate this software performance

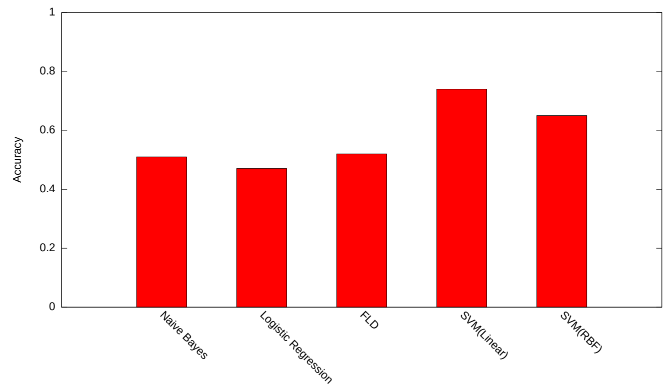

Figure 4: Comparison of accuracy for different classifiers for One Lane Bridge with noise.

anti-pattern by controlling the size of a memory pool. If the memory pool is small (or too low), then memory has to be managed using system-level calls.

We therefore created Excessive Dynamic Allocations in Apache Web Server by setting its memory pool size to 1 KB. We then used AB to send 1 million requests from 500 concurrent clients. We created the scenario without the software performance anti-pattern by setting the web server's memory pool size to 1 MB. We then used AB to send 1 million requests from 500 concurrent clients. In both scenarios, the server's concurrency level was 500 threads. Lastly, the training data set contained 400 data samples and the test data set contained 200 data samples.

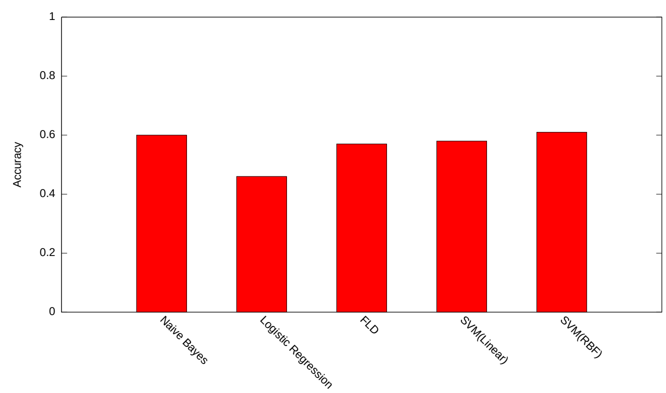

Figure 5: Comparison of accuracy for different classifiers for Excessive Dynamic Allocations.

Figure 5 shows NiPAD's accuracy results. Only Naive Bayes classifier and the SVM classifier with RBF kernel reached an accuracy level of 0.6. One reason for low accuracy results is, as shown by Berger et al. [10], custom memory allocators do not have many benefits over system-level memory allocators with respect to memory allocations of same size of objects. In our experiments, we used AB to generate requests, but the requests are generally the same. This causes the Apache Web Server to create the same size and types of objects internally to handle the requests. As a result, the two data sets do not have enough variation.

4. RELATED WORK

Parsons et al. [3] used *association rule mining* to identify performance anti-patterns in Enterprise Java Bean (EJB) applications. They use deployment descriptors, instrumentation data, and rules to automatically identify the software anti-patterns. Our approach differs two ways: (1) our approach is more general because we relax the assumption that deployment information is available; and (2) our approach is not platform- and architecture-dependent, which removes assumptions related to EJB semantics.

Bodik et al. [4] used *Logistic Regression* to classify performance crisis of data centers. Likewise, Cohen et al. [11] describes a similar approach using *Bayesian Belief Network* instead of Logistic Regression. Our work differ from their work in that we are focusing on identifying software design issues via software performance anti-patterns, and their work focuses on perform requirement violations.

Bodik et al. [12] also modeled Internet service workload spikes, such as large number of searches after the death of a popular artist, using statistical methods. They have tried to model this kind of workload using Beta distributions and Normal distributions. Unlike the proposed approach in this project, they are not using any classification techniques and their study is on special cases of software executions, not about performance metrics collected from software executions under normal conditions.

Khomh et al. [13] create BBNs from rules to find code smells, which are related to software performance anti-patterns. The BBN and original source code is then used to detect the code and design smell. Their approach, however, can be characterized as static software analysis. Our work is similar, but different, because it is characterized as a dynamic analysis technique. It is therefore hard for their approach to identify and classify software performance anti-patterns.

5. CONCLUDING REMARKS

In this paper, we formulate the software performance anti-pattern detection problem as a binary classification problem and employed machine learning classification techniques to predict unlabeled performance data. Based on our experiments and results we learned that software applications typically execute in environments with other applications, which can add noise to the data. Future work is to include application-level metrics into the analysis to help offset the impact other applications have collected data.

6. REFERENCES

[1] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2nd international workshop on Software and performance*. ACM, 2000, pp. 127–136.

[2] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *Software & Systems Modeling*, pp. 1–42, 2012.

[3] T. Parsons, "A framework for detecting performance design and deployment antipatterns in component based enterprise systems," in *Proceedings of the 2nd international doctoral symposium on Middleware*. ACM, 2005, pp. 1–5.

[4] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 111–124.

[5] R. P. Erkki Salonen, "Find the bug, Fix the bug, Do it fewer times (TimeToPic)," http://www.timetopic.net/Pages/default.aspx, 2012.

[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Symposium on Networked Systems Design and Implementation. USENIX Association*, 2012, pp. 26–26.

[7] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 145–155.

[8] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, "Unit Testing Non-functional Concerns of Component-based Distributed Systems," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, Denver, Colorado, apr 2009, pp. 406–415.

[9] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[10] E. D. Berger, B. G. Zorn, and K. S. McKinley, *Reconsidering custom memory allocation*. ACM, 2002, vol. 37, no. 11.

[11] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, vol. 6, 2004, pp. 16–16.

[12] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 241–252.

[13] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.