Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report

Tarek M. Ahmed[1], Cor-Paul Bezemer[1], Tse-Hsun Chen[1], Ahmed E. Hassan[1], Weiyi Shang[2]
Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Ontario, Canada[1]
Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada[2],
{tahmed, bezemer, tsehsun, ahmed}@cs.queensu.ca[1], shang@encs.concordia.ca[2]

ABSTRACT

Performance regressions, such as a higher CPU utilization than in the previous version of an application, are caused by software application updates that negatively affect the performance of an application. Although a plethora of mining software repository research has been done to detect such regressions, research tools are generally not readily available to practitioners. Application Performance Management (APM) tools are commonly used in practice for detecting performance issues in the field by mining operational data.

In contrast to performance regression detection tools that assume a changing code base and a stable workload, APM tools mine operational data to detect performance anomalies caused by a changing workload in an otherwise stable code base. Although APM tools are widely used in practice, no research has been done to understand 1) whether APM tools can identify performance regressions caused by code changes and 2) how well these APM tools support diagnosing the root-cause of these regressions.

In this paper, we explore if the readily accessible APM tools can help practitioners detect performance regressions. We perform a case study using three commercial (AppDynamics, New Relic and Dynatrace) and one open source (Pinpoint) APM tools. In particular, we examine the effectiveness of leveraging these APM tools in detecting and diagnosing injected performance regressions (excessive memory usage, high CPU utilization and inefficient database queries) in three open source applications. We find that APM tools can detect most of the injected performance regressions, making them good candidates to detect performance regressions in practice. However, there is a gap between mining approaches that are proposed in state-of-the-art performance regression detection research and the ones used by APM tools. In addition, APM tools lack the ability to be extended, which makes it hard to enhance them when exploring novel mining approaches for detecting performance regressions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR'16, May 14-15 2016, Austin, TX, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901774

1. INTRODUCTION

The performance of modern software applications has become a critical non-functional requirement [41]. Unfortunately, due to the increased complexity of such applications and the increased number of users, maintaining an adequate level of performance becomes challenging.

Performance regressions occur due to software updates that degrade the performance of an application. Regressions have negative consequences such as increased costs of software maintenance and user dissatisfaction [25, 30]. These regressions can even lead to substantial financial losses [40]. Amazon [7] shows that a delay of one second in page load time can decrease Amazon's sales by as much as $1.6 billion yearly. As a result, a large amount of research has focused on studying the causes of performance regressions in software applications and how to efficiently detect such regressions primarily by mining various types of operational data (such as performance counters and logs) [15, 17, 28, 33, 38, 43].

In practice, performance regression testing is performed on a software application before its release to detect performance regressions [21, 23, 31]. Performance regression testing is the process of applying a workload on two versions of a software application in order to detect whether the code changes have introduced regressions [21]. Hence, performance regression testing detects regressions when processing a stable workload due to code changes.

A plethora of research that analyzes performance data has been done to detect the performance regressions effectively. In particular, prior research in mining software repositories has shown the effectiveness of applying mining approaches to help developers identify performance regressions in large scale systems [21, 22, 27, 31, 32, 37]. However, most of such research is not easily accessible to practitioners. On the other hand, Application Performance Management (APM) tools are commonly used in practice. By integrating mining approaches on performance data into off-the-shelf performance monitoring tools, APM tools are often used to detect anomalies in the performance, instead of identifying performance regressions. Table 1 illustrates the difference between performance regression testing and APM tools. Because of the availability of mining approaches that are already integrated into APM tools and the importance of identifying performance regressions, practitioners often tend to leverage APM tools to identify performance regressions. However, APM tools are not originally designed for that task and there exists no knowledge about their effectiveness for such a task.

Table 1: Comparison between performance regression detection and APM tools

	Perf. regression detection	APM
Workload	Stable	Changing
Code base	Changing	Stable

In this paper, we investigate how suitable APM tools are for detecting performance regressions. First, we briefly highlight mining approaches used by the APM tools that make them potential candidates to be used in detecting performance regressions. Second, we study the effectiveness of APM tools in detecting performance regressions. We inject performance regressions in three open source web applications (PetClinic, CloudStore and OpenMRS). We identify these regressions using APM tools. Our study focuses on web applications because some of the studied APM tools (New Relic and Pinpoint) do not support standalone applications, and these APM tools are most commonly used for web applications.

We study the effectiveness of four APM tools, both commercial and open source, in detecting performance regressions caused by code changes. We survey the mining approaches that these APM tools use for finding and adjusting performance baselines, and identifying performance problems. We find that APM tools usually only implement very basic mining approaches, such as threshold-based anomaly detection approaches. In addition, we find that locating the root causes of the performance regressions is a time-consuming task which requires a large amount of manual work. We believe that using recently proposed software mining approaches in literature can significantly help reduce the debugging effort. Finally, we find that APM tools lack the ability to be extended, which makes it hard to enhance them when exploring novel mining approaches for detecting performance regressions.

Paper Organization. The rest of this paper is organized as follows: Section 2 presents work related to our study. Section 3 gives background information on APM tools and explains what makes them potential candidates to be used in detecting performance regressions. Section 4 describes the setup of our case study to study the effectiveness of the APM tools in detecting performance regressions. Section 5 presents the results of our case study. In Section 6, we discuss the results of our case study and our experience with using APM tools for detecting performance regressions. Section 7 discusses the threats to the validity of our case study. Finally, Section 8 concludes the paper.

2. RELATED WORK

Research on performance regression detection typically relies on mining large-scale performance data to detect performance regressions. Several recent studies report on the use of performance regression detection research in an industrial setting. Examples of the use of mining software repositories approaches for performance purposes include: Nguyen et al.'s work on the use of control charts to automatically detect performance regressions [31]. Two key benefits of their approach are the use of control charts which are simple to implement. In addition, such historical performance data are also associated with identified root-causes, in order to model and predict a possible root-cause of newly identified performance regression [32], and the use of readily available historical performance data that is archived in performance repositories. Foo et al. [22] abstract the application performance using a combination of different models that are derived from mining performance data. Each of these models corresponds to one test run. The combination of models is used to detect performance regressions. Shang et al. [37] cluster performance data and build statistical models on each cluster of performance data. Shang et al. can detect performance regressions by measuring the modeling error between two sets of performance data.

Similarly, mining software repository techniques are widely leveraged to detect performance anomalies during the monitoring of production applications. Chen et al. [16] propose an approach to mine the execution logs in order to discover the software components that are highly correlated with errors. Cohen et al. [20] use a class of probabilistic models called Tree-Augmented Bayesian Networks to identify the performance metrics that correlate with faults. Syer et al. [39] link performance data with execution logs that are generated from large software systems. Syer et al. cluster similar system scenarios using execution logs and mine the associated memory data in each cluster of scenarios to identify memory-related problems. Cherkasova et al. [19] propose a framework for automated detection of performance anomalies based on on-line modeling of the CPU demand of a software application. Maplesden et al. [28, 29] report on the use of subsuming methods to mine execution logs to detect performance optimization opportunities in a large scale production application at Netflix.

Despite of the active mining research for detecting regressions and anomalies, such research either has limited applicability or is not readily available for practitioners. The novelty of our work is the use of the readily available tools that already integrate mining approaches to detect performance regressions. This allows practitioners to use their current APM tools to detect performance regressions without the need to identify, locate, install and configure specific research tools.

3. APPLICATION PERFORMANCE MANAGEMENT

APM tools have become an essential tool for practitioners who wish to monitor large software applications in the field [19]. A plethora of commercial APM tools exist [1, 4, 8, 12]. In this section we explore the main concepts and structure of the APM tools that make them suitable for detecting performance regressions.

3.1 What is an APM Tool?

An APM tool is usually used to monitor the performance and the availability of a monitored web-based software application [42]. An APM tool collects several performance metrics (such as response time) from the monitored application and mines these metrics to measure the health of the application (e.g., identify potential performance problems using mining approaches). Most of the metrics that are mined by the APM tools are used in performance regression detection research as well. Hence, APM tools might be effective in detecting performance regressions using these metrics.

APM tools follow a typical workflow. After installation,

the APM tool discovers the different software artifacts of the application that is being monitored. This discovery step allows the APM tool to visualize the application deployment structure and to collect fine-grained metrics about the transactions that are processed by the monitored application. A transaction usually means a web request originating either from a client browser or from another application using a web service request. Examples of fine-grained metrics include: the amount of time spent in each application component for each request and the total number of transactions processed by each application component.

During the analysis phase, the APM tool collects the performance metrics periodically. The APM tool mines a historical repository of the collected metrics to determine whether a transaction is abnormal (e.g., slower than usual). APM then sends alerts to the practitioners about transactions having slow performance or issues related to the computational resources used by the monitored application.

These alerts can be categorized into three main categories:

- **Transaction-related alerts:** Information about single transactions such as the response time and the stack trace, in addition to aggregated information such as the total number of transactions and the average response time.
- **Memory-related alerts:** Information about memory usage and possible excessive memory usage.
- **Database-related alerts:** Information about the executed database queries such as query details, number of executed queries for each transaction and the time consumed to run each query.

We believe that these types of information can be used by practitioners not only to detect performance anomalies in production due to workload changes, but also to identify performance regressions (where the code is changing but the test workload is stable) across versions of a software application.

3.2 Study on APM Tools

We present an overview of three commercial APM tools: AppDynamics [1], New Relic [8] and Dynatrace [4] and one open source APM tool: Pinpoint [12], based on 1) their ease of installation and 2) their used approaches for detecting performance anomalies in production. Table 2 shows a summary of the comparison of the studied APM tools. We choose these commercial APM tools based on their popularity [26], and Pinpoint is the only mature open-source APM tool that is available today.

3.2.1 Installation

Modern large software applications are complex, as they involve different technologies and may include several programming languages. Commercial APM tools support a wide range of programming languages as illustrated in Table 3. However, the open-source APM tool (Pinpoint) supports the Java programming language only. Practitioners who wish to monitor their large applications using APM tools seek the ease of installation of such tools and the ability of these tools to discover performance anomalies in multilingual applications.

APM tools support two installation modes: cloud-based and on-premise. Cloud-based installations offer practitioners an easy installation option because practitioners do not have to manage the APM analysis and reporting components. Instead, practitioners only need to install a local agent which communicates the monitored data back to the cloud. The required total time to install the cloud-based tools is less than 30 minutes. Both New Relic and AppDynamics support cloud-based installation.

Table 3: APM tools Supported Programming Languages

	New Relic	AppDynamics	Dynatrace	Pinpoint
Java	✓	✓	✓	✓
.Net	✓	✓	✓	X
Php	✓	✓	✓	X
Ruby	✓	✓	X	X
Python	✓	✓	✓	X
Node.js	✓	✓	✓	X
C/C++	✓ (Beta)	✓ (Beta)	✓	X

On-premise installation consists of two or more components, typically an analysis component and an agent to send the data to the analysis component. The analysis component can be further divided into several components. These components can be installed separately (e.g., Pinpoint requires a separate installation of a database to store the performance data) or they can be a part of the installation package (e.g., Dynatrace).

On-premise APM tool installation is a much more elaborate process than cloud based installation. For example, we needed several hours to install the components of Dynatrace and to configure the required ports. Installation of Pinpoint required more than one day to install all the components and to troubleshoot the installation problems.

3.2.2 Application Monitoring

APM tools provide off-the-shelf support to collect the hardware metrics (e.g., Memory use, CPU utilization, I/O operations) and provide tools to visualize these metrics. The metrics are used to visualize hardware utilization and to manually specify areas of slow performance. In some situations, APM tools are unable to collect the required metrics to identify performance anomalies because the application being monitored use some frameworks that are not supported by the APM tool. Then, an additional step is required to instrument the source code manually.

Automated Instrumentation. Instrumentation is the process of adding code to a software application to monitor its behavior [24]. The studied APM tools use bytecode instrumentation in which the APM tool automatically inserts its monitoring code directly into the bytecode of the application being monitored [5, 13]. All the studied APM tools primarily monitor transactions, such as login, cart checkout and search transactions. APM tools typically collect performance metrics, such as response time and the number of called database queries, for each transaction. In addition, these metrics are aggregated to describe the overall application health. Examples of these aggregated metrics are: Average Request Size, Average Response Time, Calls per Minute, Errors per Minute, Number of Slow Calls.

Table 2: Comparison between the studied APM tools

	New Relic	AppDynamics	Dynatrace	Pinpoint
Feature	Easy	Easy	Medium	Difficult
Cloud-based installation	✓	✓	X	X
On-premise installation	X	✓	✓	✓
Custom instrumentation	✓	✓	✓	X

Table 4: Custom Instrumentation techniques in APM tools

	New Relic	AppDynamics	Dynatrace
Code-based techniques	✓	✓	✓
Configuration-based techniques	X	X	✓

Table 5: Software Monitoring Approaches

	New Relic	AppDynamics	Dynatrace	Pinpoint
Baseline-based approach	X	✓	✓	X
Threshold-based approach	✓	✓	✓	✓

Manual Instrumentation. In some cases, practitioners are allowed to manually instrument their code to obtain more information about the performance of their application. For example, if the APM tool is unable to automatically discover a specific transaction type, the practitioner can insert custom code to help the APM tool in discovering this transaction type.

APM tools provide techniques to enable custom instrumentation of the monitored application. These techniques can be categorized under two main categories [2, 6, 9]: 1) code-based techniques and 2) configuration-based techniques.

Code-based techniques require changes to the source code of the application being monitored. For example, adding annotations to a specific method or class to allow the APM to display detailed information about this specific method or class. Configuration-based techniques allow the practitioner to edit the instrumentation rules from within the APM tool interface, hence it does not require code changes.

Table 4 shows the different instrumentation techniques supported by the studied APM tools. It is worth noting that the Pinpoint APM tool does not yet offer a way to add custom instrumentation to the application being monitored.

3.2.3 Performance Analysis Approaches

The studied tools use two general mining approaches to detect performance anomalies: 1) baseline-based and 2) threshold-based. Some APM tools use only one approach (usually the threshold-based approach) such as New Relic and Pinpoint. Other APM tools support both approaches, such as AppDynamics and Dynatrace. Table 5 shows a comparison of the approaches that are used by the studied APM tools.

Baseline-based Approach. In baseline-based approaches, APM tools mine historical performance data to establish a baseline. For example, an application is expected to have a higher load during the working hours and a lower load during weekends. Hence the APM tool needs to learn this behavior. In order to detect performance anomalies, APM tools detect metric values that deviate from this baseline.

APM tools use different mining approaches to establish the baseline. For example, AppDynamics uses the average value of a metric observed during a specific time range to define the baseline. Dynatrace uses other statistical techniques such as the 90th percentile and binomial distribution to calculate the baseline. APM tools use these baselines as indicators to notify practitioners when their application performance deviates from the baselines. The specific details of these statistical techniques are not available in the documentation of the commercial tools. For example we are unable to identify the parameters of the binomial distribution used by Dynatrace.

Threshold-based Approach. A threshold is the value beyond which the performance of the monitored application is considered unacceptable. A threshold can be calculated using simple statistical methods or it can be configured either by the software practitioners or by the APM tool itself. APM tools support the following types of thresholds:

- Percentage deviation threshold: When a metric value exceeds a specific percentage above the metric's average.
- Standard deviation threshold: When a metric value exceeds multiples of standard deviations above the metric's average.
- Fixed threshold: When a metric value exceeds a predefined fixed value.

Usually, a threshold is set for the transaction response time metric because it is the APM tool's primary concern. However, some APM tools provide means to define thresholds for other metrics, such as the failure rate and throughput (in AppDynamics and Dynatrace).

Pinpoint uses fixed thresholds to decide whether the transaction is slow or not based on its response time. Specifically, Pinpoint has four thresholds for the response time of the transaction: less than one second, less than three seconds, less than five seconds, and more than five seconds. Thresholds can be manually defined by the user in all the studied APM tools except Pinpoint. In addition, some APM tools define default values for the thresholds. In the studied APM

tools, Dynatrace is the only tool that provides default values for the used thresholds.

Even though APM tools typically depend on the mining of large performance data to generate a baseline or threshold for identifying abnormal performance, these tools do not leverage the complex mining approaches that are proposed in recent research [21, 22, 27, 31, 32, 37]. The finding may imply a gap between current research and APM tools, or more complex techniques are not needed. Therefore, we are interested to explore whether such simple mining approaches that are integrated in APM tools can still effectively support the detection of performance regressions.

4. CASE STUDY SETUP

In this section we present the setup of our case study to evaluate the effectiveness of the studied APM tools in detecting performance regressions. In particular, APM tools are designed to identify slow transactions and detect unusual behavior of the monitored application. Our goal is to investigate their effectiveness in detecting performance regressions. First we present the methodology that our case study followed. We present a brief description of the used applications and finally we explain the types of injected regressions.

4.1 Methodology

Our methodology aims at detecting performance regressions and their root causes either using baseline-based or threshold-based approaches provided by the APM tools. Generally, in order to detect regressions with APM tools using a baseline-based approach, we perform the following steps:

1. Run a load test for one hour without injecting the performance regressions to define the baselines.
2. Deploy another version of the application after injecting the performance regressions.
3. Run the same load test again for one hour.
4. Explore the transaction types that show violations to the learned baselines.

To detect regressions with APM tools using a threshold-based approach, we perform the following steps:

1. Run a load test for one hour without injecting the performance regressions.
2. Explore the transactions that are marked as slow and check the methods in the stack trace of the slow transactions.
3. Deploy another version of the application with injected the performance regressions.
4. Run the same load test again for one hour.
5. Explore the transactions that are marked as slow and find those transactions that were not detected in the first step.
6. Inspect the methods in the stack traces of the detected transactions and check for the methods that were not detected before injecting the regressions.

After identifying the transaction types having performance regressions using the above-mentioned approaches, we manually drill down into the details of these transaction types and we study the information provided by the APM tool to describe each problem. Namely, we try to follow how a developer may use an APM tool for finding a performance regression. APM tools provide detailed information about each transaction such as the stack trace of the transaction and the time consumed by each method. In particular, the time consumed by each method in the stack trace is the inclusive time required by all the method callees. Therefore if we find a regression in a specific method, we check its callees until the actual cause of the regression is determined.

We try to use existing load tests[1] for our study (i.e., Pet Clinic). However, in the cases where the tests are not present (i.e., CloudStore and OpenMRS), we design and implement the tests by ourselves, and ensure that we cover all the transactions in which we inject the performance regressions. For CloudStore, we design our tests to cover common online shopping behaviours, such as browsing, searching, adding items to carts, and checking out. For OpenMRS, we design our tests to cover actions such as searching (by patient, concept, encounter, and observation), and editing/adding/retrieving patient information. We use the MySQL backup files that are provided by CloudStore and OpenMRS developers for our experiments. The backup file for Cloud Store contains data for over 5K patients and 500K observations. The backup file for Cloud Store contains about 300K customer data and 10K items.

We perform our experiments on the studied APM tools using the default configuration of each APM tool. The default configuration means that we do not perform extra effort to instrument the code or to configure the thresholds. One of our goals is to explore whether the APM tools can produce effective results out-of-the-box. In future work, we plan to explore the complexity of configuring such tools and the techniques that can be used to optimize these configurations. We describe below how we detect performance regressions by manually checking the output of each of the studied APM tools in more details, the same approach as how developers would use APM tools in practice.

New Relic. New Relic uses a threshold-based approach to detect performance anomalies. This means that New Relic does not learn baselines, rather it uses configurable thresholds to decide the overall health of the application and the health of individual transaction types.

New Relic displays the five most time consuming transaction types sorted in a descending order. For each transaction type, New Relic lists the most time consuming transactions and methods. If we find that New Relic reports a slow transaction that was not previously detected (i.e., in the initial load test), we consider that a regression was detected.

In order to identify the root cause of a detected regression, we manually inspect the stack trace of each transaction that is marked as slow by New Relic before and after injecting the regressions. We manually compare the stack traces and if we find the method in which we injected the performance problem, we consider that the root cause is identified successfully.

AppDynamics. AppDynamics uses a baseline-based approach to detect performance anomalies. AppDynamics first learns the baseline without the injected regressions. Then we deploy a new version with the injected regressions. If AppDynamics flags that the response time of a specific transaction type deviates from the baseline, we consider that the regres-

[1]https://github.com/jdubois/spring-petclinic

sion is detected.

AppDynamics records several types of information for each transaction such as the stack trace and the database queries involved in this transaction. We inspect this information in the version having the injected regressions. In case we find a transaction that is marked as slow and its stack trace or its called queries point to the injected regression, we consider that the root cause of the regression is identified correctly.

Dynatrace. Dynatrace uses a baseline-based approach to detect performance anomalies. Similar to AppDynamics, Dynatrace learns the baseline first. We follow the same approach as in AppDynamics to identify the regression and the root cause.

Pinpoint. Pinpoint uses a fixed threshold-based approach to detect performance anomalies. Pinpoint does not discover the different types of transaction types. Instead, Pinpoint lists all the transactions without grouping, which makes the process of identifying the regressions harder.

Pinpoint records the stack trace and the database queries of each transaction. Before injecting the regressions, we inspect the ten slowest transactions as recorded by Pinpoint. We record the time consumed by all methods in those transactions. Then we deploy a new version with the regressions, and repeat the previous step. If we find that the slowest transactions are different, or that the time required by a method in which we injected the regression increases, we consider that a regression is detected and the root cause is identified.

4.2 Studied Applications

In our experiments, we use three open-source applications that vary in size. The applications are PetClinic [11], CloudStore [3] and OpenMRS [10]. These applications involve different technologies. For example, PetClinic and CloudStore use heavy database transactions. OpenMRS has a RESTful web service interface that is not available in the other two applications.

PetClinic is a web application with a simple user interface to manage a pet clinic. PetClinic uses the Spring framework[2] and the Hibernate Object Relational Mapping (ORM) framework[3].

CloudStore is an open-source e-commerce web application. It is used as a benchmark application to help software developers analyze and resolve scalability related problems. CloudStore users can search for products and add products to their carts. CloudStore also uses the Hibernate ORM framework.

OpenMRS is an open-source medical records application. OpenMRS provide functionalities to manage patients and physicians in health care industry. We use the RESTful web services interface of OpenMRS to send requests to the application. Although OpenMRS has a web user interface, we decided to use the RESTful web services interface to study how the APM tools will discover the different types of RESTful web services.

4.3 Injected Performance Regressions

Performance regressions may occur due to several reasons. For example, a poorly designed database access code can lead to loading a large amount of unnecessary data [17, 18]. Earlier research study different types of code issues that cause performance regressions [17, 33, 34]. We inject performance regressions that are commonly seen in web applications, as reported by prior studies [14, 17, 32, 36]. Moreover, similar problems are known to exist in some of the studied systems [17].

For a more realistic scenario, we run the studied application with all the regressions injected at the same time. Although this might be more difficult for the APM tools to detect, it simulates an application having several performance regressions. Injecting a realistic regression requires a large amount of manual work, which involves understanding the architecture and workload of the studied systems. Thus, we are unable to inject a large number of regressions.

Below, we discuss the injected regressions cover three main areas: excessive memory usage, high CPU utilization and inefficient database use.

Excessive Memory Usage Regressions. We study two types of excessive memory usage regressions: 1) unreleased resources and 2) inefficient use of streams.

- **Unreleased resources (Memory Issue-1):** One of the well-known issues that cause an excessive memory usage problem is the unreleased resources issue [43]. Typically, this problem occurs when a developer implements a caching functionality by creating a static list and adding items to that list without ever removing them. The regression occurs because the used heap size keeps growing and the garbage collector is unable to clear the resources.
- **Inefficient use of streams (Memory Issue-2):** It occurs when a large number of streams are being opened and never closed. In this case, the garbage collector is unable to release the stream object, which leads to excessive memory usage. This issue could happen when a developer adds some code to open a stream (such as a file stream to read a file) and forgets to close this stream after reading or writing to the file is done.

High CPU Utilization Regressions. Inefficient loops usually lead to higher CPU utilization [21]. Such loops cause a delay in response time that may be detected by APM tools in addition to the CPU spikes caused at the server side. We inject two different inefficient looping problems studied recently by Nistor et al. [33, 34]: 1) unnecessary loops, and 2) loop with a missing break condition.

- **Unnecessary loops (CPU Issue-1):** This issue occurs when a method contains an inefficient nested loop. This kind of loop typically consumes the CPU which causes a long processing time. The regression occurs because the application uses more CPU resources than the previous version of the application.
- **Loop with a missing break condition (CPU Issue-2):** It occurs when the code contains a loop that should have called *break* when a specific condition is met, but instead the loop continues for a specific number of times. The regression occurs because the application uses more CPU resources than the previous version of the application.

[2]http://projects.spring.io/spring-framework/
[3]http://hibernate.org/orm/

Inefficient Database Use Regressions. We inject performance regressions that result from the inefficient use of database queries. A performance regression studied recently by Chen et al. [17] occurs due to ORM anti-patterns. In particular, two types of ORM anti-patterns are studied: 1) *Excessive Data* and 2) *One-by-one Processing*.

- **Excessive Data (Database Issue-1):** A regression that occurs because of inefficient use of eager loading in an ORM-based query. For example, in the CloudStore application, it is inefficient to load all the customer's orders on a login request. As the number of orders increases, the cost of joining *customer* and *order* tables for retrieving all the orders and their corresponding details increases. It is more efficient to load the orders only when they are needed.
- **One-by-one Processing (Database Issue-2)** is the opposite process of Excessive Data regression, where a specific field is loaded in a loop by issuing multiple queries to retrieve its value. The regression occurs if a field is originally loaded lazily but it is not used in the code, then a developer uses this field inside a loop which results in running a separate database query to load to the field during each loop iteration.

5. CASE STUDY RESULTS

In this section we present the results of our case study. We report the results for each studied APM tool. For each APM tool we report whether each injected performance regression was detected, and whether its root cause was identified. In Tables 6 and 7 we show a summary of our findings for the Pet Clinic and CloudStore applications, respectively. We do not present a similar table for OpenMRS application because the APM tools did not deal well with OpenMRS business transactions, making it difficult to obtain useful information. In the following subsections we describe our results in more detail for each studied APM tool.

5.1 Results for New Relic

PetClinic. New Relic was able to identify five out of the six injected performance regressions. The transaction with the Memory Issue-1 was identified as slower than without the injected regression. We inspected the stack trace of all the slow transactions, but the slowest methods in these transaction were not the method in which we injected the regression.

Two transactions having inefficient CPU loads were identified as slow and New Relic pointed to the correct methods with the injected regressions. We compared the stack traces of the transactions before and after injecting the regressions. We found one transaction in which the method having the regression was marked as slow. We were not able to identify this method before injecting the regression.

Finally, New Relic detected the DB Issue-1 and DB Issue-2 and showed the details of the transactions for each one. The regressions caused by both issues were detected because we compared the number of executed queries in each transaction before and after injecting the regressions.

CloudStore. Following the same methodology for the PetClinic application, New Relic was able to identify three out of five injected regressions along with the root cause method. For CPU Issue-1, we had to compare the time consumed by the methods in the slowest transactions as found by New Relic before injecting the regression. Then, we inspected the methods in the slow transactions after injecting the regression and we found that the consumed time increased for the method in which we injected the regression.

One other transaction containing a database regression (DB Issue-1) was identified as slow, however the diagnosed root cause was different from the correct root cause of the original injected regression. The diagnosed root cause was because of a pooled connection waiting issue rather than the actual injected regression. Moreover, we were able to detect DB Issue-2 by comparing the number of queries executed before and after injecting the regression which allowed us to detect the regression and identify its root cause.

OpenMRS. New Relic produced ineffective results for OpenMRS. New Relic was unable to detect the different types of RESTful services. Instead, it reported a group of GET requests and another group of POST requests. Such grouping makes it difficult for practitioners to discover performance regressions because there is no distinction between different transaction types. We tried to further analyze the slow transactions in each transaction type. For the GET requests, New Relic pointed us to another performance issue in a database query; however, this issue existed in the run without regression so we do not consider it a regression.

5.2 Results for AppDynamics

PetClinic. Using AppDynamics, we were able to detect five out of the six injected regressions. First, AppDynamics learned the baselines of all the transaction types without injecting the regressions. Then we deployed a new version containing all the regressions. AppDynamics identified the two transaction types having the CPU regressions as slow. Then we inspected the information provided by AppDynamics for the slow transactions. We find that AppDynamics provides the stack trace and the database calls for each transaction, in addition to highlighting the potential methods or database calls that are causing the slow response. We inspected the slow transactions and we found that the methods containing CPU Issue-1 and CPU Issue-2 were indicated by AppDynamics.

The transaction types having DB Issue-1 and DB Issue-2 were also identified as deviating from the baseline. The database queries used in each of the database regressions were highlighted by AppDynamics as potential problems so we considered that AppDynamics identified the root cause of the database regressions.

CloudStore. AppDynamics showed the transaction types with the performance regressions as the slowest transaction types. For each type, we drilled down into each individual slow transaction to explore the root cause of its slow response. The root cause diagnosis of the slowest transaction was usually a database connection issue. We were not able to identify the methods with the performance regressions in three out of six transaction types.

However, AppDynamics was able to identify the slow method involving CPU Issue-1. Additionally AppDynamics was able to detect the slow query in DB Issue-1 and the large number of queries in DB Issue-2.

Table 6: Pet Clinic Results

	New Relic		AppDynamics		Dynatrace		Pinpoint	
	Identified	Root Cause	Identified	Root Cause	Identified	Root Cause	Identified	Root Cause
Memory Issue-1	✓	X	✓	✓	✓	✓	X	X
Memory Issue-2	X	X	X	X	X	X	X	X
CPU Issue-1	✓	✓	✓	✓	✓	✓	X	X
CPU Issue-2	✓	✓	✓	✓	✓	✓	X	X
DB Issue-1	✓	✓	✓	✓	✓	✓	✓	✓
DB Issue-2	✓	✓	✓	✓	✓	✓	X	X

Table 7: CloudStore Results

	New Relic		AppDynamics		Dynatrace		Pinpoint	
	Identified	Root Cause	Identified	Root Cause	Identified	Root Cause	Identified	Root Cause
Memory Issue-1	✓	✓	✓	X	✓	✓	X	X
Memory Issue-2	X	X	✓	X	X	X	X	X
CPU Issue-1	✓	✓	✓	✓	✓	✓	X	X
CPU Issue-2	X	X	X	X	X	X	X	X
DB Issue-1	✓	X	✓	✓	✓	✓	✓	✓
DB Issue-2	✓	✓	✓	✓	✓	✓	X	X

OpenMRS. The different types of RESTful services were not detected automatically by AppDynamics using its default configuration, rather, the different web services need to be configured manually. This configuration is time-consuming for large applications and requires business knowledge of the monitored application.

AppDynamics grouped all the transaction types under one type only. We further analyzed the identified transaction type, we were able to detect only the DB Issue-1 using AppDynamics, because one of the transactions having this regression was identified as slow. This transaction allowed us to explore the issue's details and to identify the regression.

5.3 Results for Dynatrace

Pet Clinic. We were able to identify five out of six regressions. By analyzing the memory snapshots captured by Dynatrace, we found the growing list in Memory Issue-1 after injecting the regression although it did not exist before injecting the regressions. The transaction types having CPU Issue-1 and CPU Issue-2 were identified as deviating from the baseline. Dynatrace provided information about the CPU time consumed by each method in a transaction. From the CPU time provided by Dynatrace, we calculated that the average CPU usage increased in the CPU Issue-1 by 14% and in CPU Issue-2 by 7%. Hence, we considered that Dynatrace detected and identified the root cause of both CPU regressions.

Both DB Issue-1 and DB Issue-2 were identified as regressions because the response time of the corresponding transaction types deviated from the baseline. Dynatrace provides information about the number of executed queries and the time spent to execute each query. Dynatrace was able to identify that a database query is consuming a long time (DB Issue-1) and that a large number of queries were executed (DB Issue-2).

CloudStore. Dynatrace detected the high heap usage and automatically created a memory snapshot. We were able to identify the Java class that caused the excessive memory usage regression (Memory Issue-1). However, we were not able to identify Memory Issue-2. Additionally, we were able to identify CPU Issue-1 that caused a CPU regression. Finally, we compared the database reports to detect database regressions. We were able to conclude that DB Issue-1 exists by examining the difference in the query running time before and after injecting the regression. DB Issue-2 was identified by Dynatrace because it detected the large number of executed queries, which did not exist before injecting the regression.

OpenMRS. Dynatrace was not able to detect the different types of services automatically. Rather, splitting the different types of services needs to be done manually by the user. Using our approach to detect regressions, we were only able to identify DB Issue-2.

Pinpoint. Pinpoint uses a fixed threshold approach, which depends on reporting the transactions times and identified the slow ones if they exceed a fixed threshold. Fixed thresholds are known to be less flexible and do not fit the complex nature of modern software applications [35]. We were able to identify DB issue-1 in Pet Clinic and DB Issue-1 in CloudStore. The other regressions were not detected by Pinpoint. The results of Pinpoint and the lack of open source APM tools show that there is still much room of improvement for the open source community to create and improve such open source APM tools by integrating exiting research techniques into practice.

Commercial APM tools are good at detecting performance regressions that cause a delay in the response time; while open source APM tool (Pinpoint) is less flexible and effective in detecting performance regressions. Using all the studied APM tools, we were able to detect five out of six regressions in the Pet Clinic application and four out of six regressions in CloudStore. However, we cannot detect many performance regressions in OpenMRS.

6. DISCUSSION

In this section we present a discussion on our experience in detecting performance regressions using APM tools.

6.1 More Mining Approaches are Needed for Reducing Manual Effort

The effort required to identify the performance regressions varies according to the features provided by each APM tool. The easiest APM tool to detect the regressions is AppDynamics because it lists all the transaction details and the corresponding executed database queries. However, using these features, we still spent about six hours to detect the six regressions and identify their root causes in each studied application. The other APM tools require even more effort because their features are either too simplified such as Pinpoint, or too complicated such as Dynatrace. For Dynatrace, we spent about two days for each studied application to identify the root cause of the injected regressions. Note that we did not have prior knowledge about the APM tools.

We believe that leveraging more complex mining approaches can significantly reduce the manual effort for identifying performance regressions. For example, some APM tools rank transactions based on their absolute response time. However, the performance impact of some performance regressions may not be large enough to be flagged by the APM tools. As a system evolves, such performance regressions may have larger impacts and become more difficult to fix. Instead, if the APM tools can apply mining approaches such as statistical modelling to identify abrupt changes in the response time for a specific transaction, practitioners can locate the performance regressions more easily.

6.2 APM Tools should Provide Better Data Aggregation and Summarization

In this subsection, we provide our experience on using the data presented by APM tools for identifying performance regressions. In particular, we focus on three categories of data: 1) transaction-related, 2) memory-related, and 3) DB-related information.

Transaction-related Information. We use transaction related information to detect CPU regressions, since CPU regressions cause a delay in the response time. Our results show that the APMs are able to detect the transaction types with CPU regressions effectively. However, we also find that all the APM tools, except Dynatrace, tend to show the amount of processing time in each method as a bulk value or a percentage of the total processing time, regardless the amount of CPU, I/O, or waiting time in each method. Dynatrace shows the percentage of CPU, I/O and waiting time spent in each method, which can be used to identify methods of increasing CPU times. We were able to detect the regressions in PetClinic application, however the regressions were not detected in CloudStore.

APM tools need more fine-grained information, such as the amount of CPU, I/O, or waiting time, about individual method calls.

Memory-related Information. Although excessive memory usage is a common performance regression and all studied APM tool supports diagnostic approaches for memory-related problems, we find that memory related information needs more manual work to view and analyze than other types of information. Since creating memory snapshots is an expensive operation, APM tools tend to minimize or even ignore this type of information. For example, Memory Issue-1 required creating manual snapshots in AppDynamics to be detected. On the other hand, Dynatrace automatically detected the increased use of heap memory and created the snapshot, which allowed us to easily detect the regression. The open streams memory regression (Memory Issue-2) was not detected by any APM tool, although we believe that such a linear increase in the created objects in the heap must be identified as an issue by the APM tool. In short, sampling memory usage and identify upward trends using data analysis can significantly help practitioners identify memory-related performance regressions.

APM tools need more complex analysis approaches in order to detect memory regressions more effectively.

Database-related Information. All APM tools report detailed information about the used database queries and execution times. However, all APM tools, except New Relic, deal with the queries as a black box. This means that although the APM tools can detect the exact executed queries, and their corresponding counts and execution times, these tools do not show details of the execution plan of the query, or the used indices. New Relic, on the other hand, provides this functionality in its main APM tool by suggesting changes to the executed query to optimize its performance. Pinpoint shows the least amount of database information with each database regression. For example, in DB Issues-1, although there is a large number of executed queries, Pinpoint does not split the types of executed queries (such as SELECT and UPDATE). Rather, it only displays the total number of executed queries for this specific transaction.

APM tools provide detailed DB-related information such as the exact executed queries. However, only New Relic provides suggestions to optimize slow queries.

6.3 APM Tools Lack Flexibility for Adding New Mining Approaches

Based on our experience and the results of our experiments, we were able to detect most of the performance regressions we injected using current APM tools. The process of detecting performance regressions was relatively simple, however the process of identifying the method that contains

the regressions was more challenging. Using our methodology to detect performance regressions (based on baseline-based and threshold-based approaches), APM tools may detect a large number of slow transactions, some of which are not directly related to the injected performance regression. For example, if the load on the database is high, then some transactions will spend a long time waiting to obtain a database connection. For this reason, we had to explore a large number of transactions to identify the root cause of the injected regression.

We find that APM tools lack the ability to be extended using custom techniques that may make the process of finding the root cause (i.e. the method having the regression) easier. APM tools need more effective techniques for summarization and filtration of their results. Prior research by Nguyen et al. [32] leverages mining historical performance data with known root-causes in order to automatically identify root-causes of newly identified performance regressions. APM tools can adopt such techniques to improve the identification of root causes. Summarization means that the APM tools show a summary of the metrics and methods that contain performance regressions. Control charts proposed by Nguyen et al. [31] could be used to summarize the collected metrics to show only the metrics that deviate from a specified threshold. Subsuming method analysis proposed by Maplesden et al. [28] could be used to summarize the methods that contain performance regressions only. The current APM tools show the complete stack trace of each transaction.

Filtration means that the APM tool narrows down the results to include only the relevant performance regressions rather than showing all the performance anomalies to practitioners. One way to achieve this is by mining code change histories (such as details about modified, deleted or added code). Hence APM tools can indicate the code that is responsible for the regression by only looking at the modified code.

7. THREATS TO VALIDITY

In this section we discuss the threats to the validity of our study. Although APM tools support several programming languages, our study focuses on the Java programming language. We believe this study can be extended to other programming languages. This study does not essentially present a full functionality review of the APM tools, rather, it presents the basic ideas behind some of the many functions provided by such well-founded APM tools.

The injected performance regressions are based on some of the common regressions known in literature and in practice. We try to keep the injected performance regressions simple so that they will not introduce any functional impacts on the studied systems; however, the implementation of such regressions may vary from real life situations. We did not consider other types of performance regressions from literature. These types of regressions may be included in future work.

Our discussion of the APM tools is based solely on our experience, and may be subjective. In the future, we plan to conduct a user study regarding user experience of these APM tools. We also plan to compare advanced algorithms for detecting performance regressions with the approached used in current APM tools.

We perform our case study on four APM tools based on their popularity. However, there may not be a relationship between a tool's popularity and its effectiveness. Other available APM tools may have the ability to detect the injected regressions more efficiently. Additionally, we use only three open-source applications, hence our results may not generalize to other applications.

8. CONCLUSIONS

We explore a novel use of APM tools to detect performance regressions caused by code changes across the same workload. Although APM tools were primarily designed to leverage the mining of large performance data to detect performance anomalies in stable code base due to workload changes, APM tools were effective in detecting performance regressions caused by code changes using a stable workload. Our results show that APM tools can detect more than 75% of the injected regressions on average. This percentage dropped in applications using RESTful web services to less than 20%.

APM tools could detect most of the regressions that we injected, however, the process of identifying the root cause of the regression (i.e. the exact method causing the regression) was more challenging. APM tools provide the stack traces for a large number of transactions, many of which are unrelated. Therefore, we had to explore a large number of transactions to identify the injected regressions.

We consider APM tools strong candidates for detecting performance regressions due to their availability for practitioners and their out-of-the-box regression detection capability. However, there exists a gap between the mining approaches that are integrated in APM tools and the mining approaches that are leveraged in performance regression detection research. We believe that future development of APM tools should be focused on improving the extensibility of these tools, so that researchers can extend these tools with their state-of-the-art approaches. In addition, the reporting capability of APM tools must be improved to reduce the effort that is required to analyze detected performance regressions.

References

[1] Appdynamics. https://www.appdynamics.com/. Last accessed Sept 8 2015.

[2] Appdynamics custom instrumentation. https://docs.appdynamics.com/display/PRO41/Monitor+JMX+MBeans. Last accessed Sept 23 2015.

[3] Cloud store. https://github.com/CloudScale-Project/CloudStore. Last accessed Sept 8 2015.

[4] Dynatrace. http://www.dynatrace.com/. Last accessed Sept 8 2015.

[5] Dynatrace instrumentation. https://community.dynatrace.com/community/display/DOCDT61/Instrumentation. Last accessed Oct 16 2015.

[6] Dynatrace PurePath. https://community.dynatrace.com/community/display/DOCDT60/PurePath+Explained. Last accessed Sept 24 2015.

[7] How one second could cost amazon $1.6 billion in sales. http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales. Last accessed Oct 4 2015.

[8] New Relic. http://newrelic.com/. Last accessed Sept 9 2015.

[9] New Relic custom instrumentation. https://docs.newrelic.com/docs/agents/java-agent/custom-instrumentation/java-custom-instrumentation. Last accessed Sept 24 2015.

[10] Open MRS. http://openmrs.org/. Last accessed Sept 8 2015.

[11] Pet clinic. https://github.com/spring-projects/spring-petclinic. Last accessed Sept 8 2015.

[12] Pinpoint. https://github.com/naver/pinpoint. Last accessed Sept 8 2015.

[13] Technical overview of pinpoint. https://github.com/naver/pinpoint/wiki/Technical-Overview-Of-Pinpoint. Last accessed Oct 16 2015.

[14] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 739–753, 2010.

[15] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing I/O performance regressions. *Journal of Software: Evolution and Process*, 26(12):1193–1212, 2014.

[16] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of International Conference on Dependable Systems and Networks, DSN'02.*, pages 595–604, 2002.

[17] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering, ICSE*, pages 1001–1012. ACM, 2014.

[18] T.-H. Chen, S. Weiyi, A. E. Hassan, M. Nasser, and P. Flora. Detecting problems in database access code of large scale systems - an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[19] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Automated anomaly detection and performance modeling of enterprise applications. *ACM Transactions on Computer Systems*, 27(3):6:1–6:32, Nov. 2009.

[20] 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 - Volume 6, OSDI'04*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[21] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of 10th International Conference on Quality Software, QSIC'10*, pages 32–41, July 2010.

[22] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 159–168, 2015.

[23] S. Ghaith, M. Wang, P. Perry, and J. Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *Proceedings of 17th European Conference on Software Maintenance and Reengineering, CSMR'13*, pages 379–383, March 2013.

[24] T. Kempf, K. Karuri, and L. Gao. *Software Instrumentation*. John Wiley & Sons, Inc., 2007.

[25] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Proceedings of 10th IEEE/IFIP Network Operations and Management Symposium, 2006. NOMS'06.*, pages 373–381, April 2006.

[26] J. Kowall and W. Cappelli. Magic quadrant for application performance monitoring, October 2014.

[27] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1012–1021, 2013.

[28] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy. Subsuming methods: Finding new optimisation opportunities in object-oriented software. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 175–186. ACM, 2015.

[29] D. Maplesden, K. von Randow, E. Tempero, J. Hosking, and J. Grundy. Performance analysis using subsuming methods: An industrial case study. In *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE'15*, volume 2, pages 149–158, May 2015.

[30] T. Nguyen. Using control charts for detecting and understanding performance regressions in large software. In *IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST'12*, pages 491–494, April 2012.

[31] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE'12*, pages 299–310. ACM, 2012.

[32] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 232–241, 2014.

[33] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE'15*, pages 902–912, 2015.

[34] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE'13*, pages 562–571, 2013.

[35] A. Oliveira. Why static thresholds don't work. http://info.prelert.com/blog/623. Last accessed Sept 29 2015.

[36] S. Pertet and P. Narasimhan. Causes of failure in web applications. Carnegie Mellon University, Technical Report, CMUPDL-05-109.

[37] W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Au-

tomated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 15–26, 2015.

[38] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd International Workshop on Software and Performance, WOSP'00*, pages 127–136, 2000.

[39] M. Syer, Z. M. Jiang, M. Nagappan, A. Hassan, M. Nasser, and P. Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM), 2013 29th*, pages 110–119, 2013.

[40] T. J. Veasey and S. J. Dodson. Anomaly detection in application performance monitoring data. In *Proceedings of International Conference on Machine Learning and Computing (ICMLC)*, pages 120–126, 2014.

[41] J. Waller. *Performance Benchmarking of Application Monitoring Frameworks*. BoD–Books on Demand, 2015.

[42] C. Wang, L. Su, X. Zhao, and Y. Zhang. Application performance monitoring and analyzing based on bayesian network. In *Proceedings of the 11th Web Information System and Application Conference, WISA,14*, pages 61–64, Sept 2014.

[43] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of ACM/IEEE 30th International Conference on Software Engineering, ICSE'08*, pages 151–160, 2008.

View publication stats