Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring

Kundi Yao, Guilherme B. de Pádua, Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
{ku_yao,g_bicalh,shang}@encs.concordia.ca

Steve Sporea, Andrei Toma, Sarah Sajedi
ERA Environmental Management Solutions
Montreal, Quebec, Canada

ABSTRACT

Performance assurance activities are an essential step in the release cycle of software systems. Logs have become one of the most important sources of information that is used to monitor, understand and improve software performance. However, developers often face the challenge of making logging decisions, i.e., neither logging too little and logging too much is desirable. Although prior research has proposed techniques to assist in logging decisions, those automated logging guidance techniques are rather general, without considering a particular goal, such as monitoring software performance. In this paper, we present Log4Perf, an automated approach that provides suggestions of where to insert logging statement with the goal of monitoring web-based systems' software performance. In particular, our approach builds and manipulates a statistical performance model to identify the locations in the source code that statistically significantly influences software performance. To evaluate Log4Perf, we conduct case studies on open source system, i.e., CloudStore and OpenMRS, and one large-scale commercial system. Our evaluation results show that Log4Perf can build well-fit statistical performance models, indicating that such models can be leveraged to investigate the influence of locations in the source code on performance. Also, the suggested logging locations are often small and simple methods that do not have logging statements and that are not performance hotspots, making our approach an ideal complement to traditional approaches that are based on software metrics or performance hotspots. Log4Perf is integrated into the release engineering process of the commercial software to provide logging suggestions on a regular basis.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Software performance**;

KEYWORDS

Software logs; Logging suggestion; Performance monitoring; Performance modeling; Performance Engineering

ACM Reference Format:
Kundi Yao, Guilherme B. de Pádua, Weiyi Shang and Steve Sporea, Andrei Toma, Sarah Sajedi. 2018. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184416

1 INTRODUCTION

The rise of large-scale software systems, such as web-based system like Amazon, has imposed an impact on people's daily lives from mobile devices users to space station operators. The increasing importance and complexity of such systems make their quality a critical, yet a hard issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs [39]. Therefore, performance assurance activities are an essential step in the release cycle of large software systems.

Monitoring performance of large systems is a crucial task of performance assurance activities. In practice, performance data is often collected either based on system-level information [17], such as collecting CPU usage, or application-level information, such as response time or throughput. In particular, Application Performance Management tools, such as Kieker [38], are widely used in practice. They collect performance data from the systems when they are running in the field environment. However, such system or application-level performance data often leads to the challenges of pinpointing the exact location in the source code that is related to performance issues.

On the other hand, the knowledge of logs has been widely identified to improve the quality of large software systems [14, 15, 24, 25, 33]. Prior research proposed and used logs to monitor and improve software performance [14, 15, 24, 33]. The success of those performance assurance techniques depends on the well-maintained logging infrastructure and the high quality of the logs. Although prior research has proposed various approaches to improve the quality of logs [21, 27, 28, 42, 43, 46, 47], all of these approaches consider logs in general, i.e., not considering the particular need of using logs for performance assurance activities. Therefore, the suggested improvement of logs may not be of interest in performance assurance activities.

In this paper, we present an approach that automatically provides logging location suggestion for web-based systems based on

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 ACM 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.
ICPE'18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184416

the particular interest in performance modeling. Our approach first automatically insert logging statements into the source code. After conducting performance tests with the system, our approach builds statistical performance models to represent the system performance (such as CPU usage) using logs that are generated by the automatically inserted logging statements in the source code. By improving and analyzing statistical performance models, our approach identifies the logging statements that are statistically significant in explaining the system performance. Such logging statements are suggested to practitioners as potential logging locations for the use of performance assurance activities.

We evaluate our approach with two open source systems, namely OpenMRS and CloudStore, and one commercial system. Our evaluation results show that we can build high-quality statistical performance models with R^2 between 26.9% and 90.2%. By studying the suggested logging locations, we find that they all have a high influence on the system performance. Also, these locations cannot be identified using code complexity metrics or detected as performance hotspots. This paper makes the following contributions:

- To the best of our knowledge, our work is the first to provide logging suggestions with the particular goal of performance monitoring.
- We propose a statistically rigorous approach to identifying source code locations that can statistically explain system performance.
- The outcome of our approach can complement the use of traditional code metrics and performance hotspots to assist performance engineers in practice.

Our approach is already adopted in an industrial environment and is integrated into a continuous deployment environment. Developers receive logging suggestions from our automated approach regularly to better monitor the system performance in the field.

The rest of this paper is organized as follows: Section 3 presents our automated approach to suggest logging locations. Section 4 and 5 present the results of evaluating our approach through answering three research questions and discuss related topics based on the results. Section 6 presents the prior research that is related to this project. Section 7 presents the threats to the validity of our study. Finally, Section 8 concludes this paper.

2 A MOTIVATING EXAMPLE

Tom is a performance engineering of an e-commerce web system. He often uses the information from web logs (e.g., page requests) to build performance models to understand system performance or to detect performance issues.

Tom finds that the performance models are often unreliable in predicting the system's performance. He examines the performance of each log entry and found that some entries have a significant variance. However, there is not enough information in the web logs to accurately pinpoint the issue for further monitoring. Hence, by knowing only which web requests were called is not enough to explain the performance of the system.

Let us consider an example (Algorithm 1) in which the function process a list of products of a given signed-in customer. The products have an expiration date and, if they are expired, the program needs to consult a different supplier. In this example, the method *LoadProductStock* response time varies according to different factors, such as the number of products for that customer, and whether the products are expired or not. If the products are not expired the method might return very fast; while if the current customer has many expired products, there will be too many calls to consult suppliers, leading to the significantly long response time.

Algorithm 1 Example: Load products that has an expiration date.

```
1: function LoadProductStock(c)
2:     products ← product list of customer c
3:     for each p in products do
4:         if IsExpired(p) then
5:             p.Stock ← StockFromSupplier(p)
6:         end if
7:     end for
8: end function
```

Although Tom can identify and monitor some complex requests in the web logs, he finds that some complex requests may not be so useful to monitor, since they have a steady performance behavior. For those cases, the information provided by the web logs is sufficient. Nevertheless, for the requests that their performance is not steady (e.g., Algorithm 1), there exists a high degree of uncertainty. Due to this reason, Tom needs to manually go through all the web log entries to find the scenarios (e.g., particular customer and product(s)) that required further monitoring and, therefore, require more logging statements. For a large-scale system with a non-trivial workload, this manual operation is not feasible, and, consequently, Tom needs a technique to automatically suggest where the monitoring and logging are needed, without repetitive information. Such technique would significantly reduce the uncertainty of monitoring or not the right places.

In this next section of this paper, we will present an approach that seeks to suggest logging locations by examining whether the location in the source code can provide significant explanatory power to the systems' performance.

3 APPROACH

In this section, we present our approach that can automatically suggest logging locations for software performance monitoring. To reduce the performance overhead caused by introducing instrumentation into the source code, we first leverage the readily available web logs to build a statistical performance model, and we identify the web requests that are statistically significantly performance-influencing. In the second step, we only focus on the methods that are associated with the performance-influencing web requests and identify which method is statistically significantly performance-influencing. Finally, we focus on the basic block in the source code that is associated with the performance-influencing method, and we identify and suggest the code blocks that logs should be inserted.

For each step, we apply a workload on the subject system while monitoring its performance. Afterwards, we build a statistical model for the performance of the subject system using either the readily available web logs or the automatically generated logs from

instrumentation during the workload. Using the statistical performance model, we identify the statistically significant performance-influencing logging statements. The overview of our approach is presented in Figure 1.

Step 1: Identifying performance-influencing web requests

In the first step, we aim to identify the source code associated with the web requests that influence system performance.

1.1 Parsing web logs

We run performance test for our subject systems and monitor their performance during the test. After the performance test, we parse the generated web logs. In particular, we keep the time stamp of the web log and the web request (e.g., a restful web request).

We then calculate log metrics based on those logs. Each value of each log metric L is the number of times that each web request executes during the period. For example, if a web request *index.jsp* is executed 10 times during a 30-second time period, the metric *index.jsp*'s value is 10 for that period.

1.2 Building statistical performance models using web logs

We follow a model building approach that is similar to the approach from prior software performance research [17, 33, 40]. We build a linear regression model [20] to model the performance of the software. We choose linear regression model because: 1) the goal of the approach is not to build a perfect model but to interpret the model easily instead, and 2) prior research used such modeling techniques to model software performance [17, 18, 40]. We use the log metrics that are generated from web logs as independent variables. The dependent variable of the model is the performance metrics that are collected during applying the load on the software system, such as CPU usage.

After building a linear regression model for the performance of the software, we examine each independent variable, i.e., log metric, to see how statistically significant it is to the model's output, i.e., performance metrics. In particular, we only consider the log metrics that have p-value ≤ 0.05. Since each log metric represents the number of times that the associated source code of each web request executes, the significance of a log metric shows whether the execution of the web log associated source code has a statistically significant influence on the software performance. Based on the list of statistically significant log metrics, we identify the performance-influencing web requests.

Step 2: Identifying performance-influencing methods

In the second step, we focus only on the performance-influencing web requests, and we aim to identify which methods in the source code are statistically significantly influencing performance. To reduce the performance overhead of the instrumentation, we note that every time we only focus on *one* performance-influencing web request. If multiple web requests are found performance-influencing, we repeat this step for every one of them.

2.1 Automatically inserting logging statements into methods

In this step, we automatically insert a logging statement into every method that is associated with the performance-influencing web requests. We use source code analysis frameworks, such as Eclipse JDT [2] and .NET Compiler Platform ("Roslyn") [5], to parse the source code and to identify the associated methods in the source code. We automatically insert a logging statement based on *Log4j2* and *Log4Net.Async* at the beginning of each method source code. Since the goal of our approach only suggests the location to insert logging statement, we only print the time stamp and the method name using the logging statement. After re-building the systems and applying performance tests to each subject system, logs will be generated automatically.

Similar to step 1.1, we parse both the web logs and the logs that are generated by our inserted logging statement. Then we generate log metrics based on these logs.

2.2 Reducing metrics

Intuitively, methods that never execute during a workload, or the execution of the method has a constant value during the workload do not influence the performance of the system. Hence, we first remove any log metric that has constant values in the dataset. Methods may often be called together, or one method may always call another one. In such cases, not all methods need to be logged. Hence, we perform a correlation analysis on the log metrics [26]. We used the Pearson correlation coefficient among all performance metrics from one environment. We find the pair of log metrics that have a correlation value higher than 0.9. From these two log metrics, we remove the metric that has a higher average correlation with all other metrics. We repeat this step until there exists no correlation higher than 0.9.

We then perform redundancy analysis on the log metrics. The redundancy analysis would consider a log metric redundant if it can be predicted from a combination of other metrics [23]. We use each log metric as a dependent variable and use the rest of the log metrics as independent variables to build multiple regression models. We calculate the R^2 of each model and if the R^2 is larger than a threshold (0.9), the current dependent variable (i.e., log metric) is considered redundant. We then remove the performance metric with the highest R^2 and repeat the process until no log metric can be predicted with R^2 higher than the threshold. For example, if method *foo* can be linearly modeled by the rest of the performance metrics with $R^2>0.9$, we remove the metric for method *foo*.

2.3 Building statistical performance models using both web logs and our generated logs

In this step, we build a similar statistical model as step 1.2. As a difference, we do not include the log metrics from web logs that are found not performance influencing from step 1.2. We follow the same model building process and the same way of identifying statistically significant log metrics. The outcome of this step is the methods that are statistically significantly performance-influencing.

Step 3: Identifying performance-influencing basic code blocks

A method may be long and consist of many basic blocks. It may be the case that only a small number of basic blocks are performance-influencing. Therefore, in the final step, we focus only on the

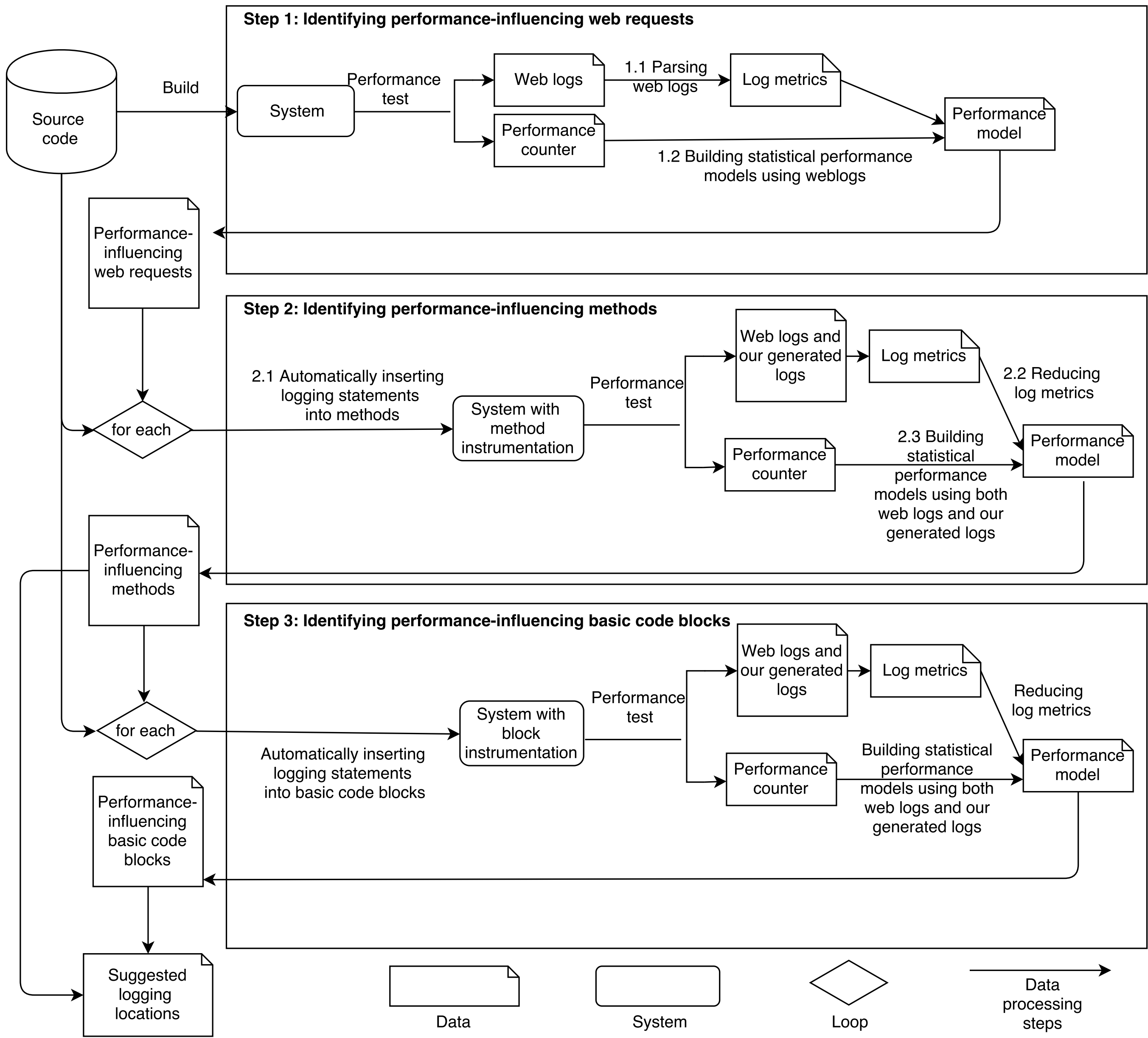

Figure 1: An overview of our approach. The numbered steps in the figure correspond to the steps in Section 3.

performance-influencing methods, and we aim to identify which basic code block is performance-influencing. Similarly, every time we only focus on one method. If multiple methods are found performance-influencing, we repeat this step for each method.

We use the code analysis frameworks to identify basic blocks of each performance-influencing methods. If a performance influencing method only contains one basic block, we do not proceed with this step. For the methods with multiple basic blocks, similar to step 2.1, we automatically insert logging statement into every basic block and generate log metrics by both the web logs and our generated logs. We also follow a similar approach as step 2.2 and 2.3 to identify which code block is statistically significantly influencing performance. We then automatically suggest to developers the logging statement insertions into the basic code block, to assist in performance monitoring. If none of the log metrics that are based on basic blocks are significant, we suggest to developers the direct insertion of logging statement at the beginning of the method itself.

4 EVALUATION

In this section, we first present the setup of our evaluation, including the subject systems, the workload, and the experimental environment. Then we evaluate our approach by answering three

research questions. For each research question, we present the motivation for the question, the approach that we use to answer the question and finally the results.

4.1 Subject systems and their workload

We evaluate our approach with open-source software, including *OpenMRS* and *CloudStore*, and one commercial software, *ES*. The overview of the subject software is shown in Table 1.

Table 1: Overview of our subject systems.

Subjects	Version	SLOC (K)	# files	# methods
CloudStore	v2	7.7	98	995
OpenMRS	2.0.5	67.3	772	8,361
ES	2017	>2,000	>9,000	>100,000

4.1.1 OpenMRS. *OpenMRS* is an open-source patient-based medical record system commonly used in developing countries. *OpenMRS* is built by an open community that aims to improve healthcare delivery through a robust, scalable, user-driven, open source medical record system platform. Their application design is customizable with low programming requirements, using a core application with extendable modules. We choose *OpenMRS* since it is highly concerned with scalability and its performance has been studied in prior research [13]. *OpenMRS* provides a web-based interface and RESTFul services. We deployed the *OpenMRS* version 2.0.5 and the data used are from MySQL backup files that are provided by *OpenMRS* developers. The backup file contains data for over 5K patients and 476K observations. We use the RESTFul API test cases created by Chen et al. [13]. The tests are composed of various searches, such as: by patient, concept, encounter, and observation, and editing/adding/retrieving patient information. The tests include randomness to simulate real-world workloads better. We keep the workload running for five hours. To minimize the noise from the system warmup and cool-down periods, we do not include the data from the first and last half an hour of running the workload. In the end, we keep four hours of data from each performance test.

4.1.2 CloudStore. *CloudStore* is an open-source sample e-commerce web application developed to be used for the analysis of cloud characteristics of systems, such as capacity, scalability, elasticity, and efficiency. It follows the functional requirements defined by the TPC-W standard for verifiable transaction processing and database benchmarks data [7]. It was developed to validate the European Union funded project called CloudScale [1]. We choose *CloudStore* due to its importance in improving cloud systems performance and scalability. It has also been studied in prior research [13]. We deployed the *CloudStore* version v2 and the data used was generated using scripts provided by *CloudStore* developers. The generated data for CloudStore contains about 864K customers, 777K orders, and 300 items. We use the test cases created by Chen et al [13] to cover searching, browsing, adding items to shopping carts, and checking out. The tests include randomness to simulate real-world workloads better. For example, there is randomness to ensure that some customers may check out, and some may not. We run the performance tests with the same length as OpenMRS.

4.1.3 ES. *ES* is a commercial software that provides government-regulation related reporting services. The service is widely used as the market leader of its domain. Because of a non-disclosure agreement, we cannot reveal additional details about the system. We do note that it has over ten years of history with more than two million lines of code that are based on Microsoft .Net. We run a typical loading testing suite as the workload of the system.

4.2 Experimental environment

The experimental environment for the open-source software is set up on three separate machines. The first machine is the database server; the second is the web server in which the web application was deployed and, finally, the third machine simulates users using the JMeter load driver [11]. These machines have the same hardware configuration, which is 8G of RAM and Intel Core i5-4690 @ 3.5 GHz quad-core CPU. They all run the Linux operating system and are connected to a local network.

We use *PSUtil* [6] to monitor the performance of the software. To minimize the noise of other background processes, we only monitor the process of the subject system that is under the workload. We monitor the CPU usage during the workload for every 10 seconds. In particular, similar to prior research [10, 37], CPU percentage of the monitored process between two timestamps are calculated as the CPU usage of the corresponding workload during the period.

The experimental environment for *ES* is an internal dedicated performance testing environment, also with three machines. The testing environment is deployed with performance monitoring infrastructure. Similar to the open-source software, we monitor the CPU usage of the process of *ES* for every 10-seconds and use a logging library to generate automatically instrumented logs.

To combine the two datasets of performance metrics and logs, and to further reduce the impact of recording noises, we calculate the mean values of the performance metrics in every 30 seconds. Then, we combine the datasets of performance metrics and system throughput based on the time stamp on a per 30-seconds basis. A similar approach has been applied to address mining performance metrics challenges [19]. We use *Log4J2*'s asynchronous logging to generate the automatically instrumented logs since it is shown to have the smallest performance overhead [4].

RQ1: How well can we model system performance?

Motivation.

The success of our approach depends on the ability to build a well fit statistical models for software performance. If the models built by our approach are of low quality, we cannot use such models to understand the influence of logged source code locations (i.e., log metrics) to the software performance (i.e., performance metrics). Additionally, the automatically inserted logging statements have an impact on software performance. If the performance is influenced by those inserted logging statements, instead of the existing source code itself, our model cannot be used to identify performance-influencing source code locations to log.

Furthermore, if we identify too many locations that are statistically significantly influencing performance, it is not practical for developers to log all locations nor can developers deeply investigate

every location to ensure the need for logging. Besides, if all the identified locations are already well logged, developers may not need our approach's logging suggestion.

Approach.

We measure the model fit to assess the quality of the statistical models for software performance. In particular, we calculate the R^2 of each model to measure model fit. If the model perfectly fits the data, the R^2 of the model is 1, while a zero R^2 value indicates that the model does not explain the variability of the dependent variable (i.e., performance metric). We also count the number of logging locations that are suggested by our approach. For every suggested logging location, we manually examine whether there already exists a logging statement.

Results.

Our model can well explain system performance. Shown by Table 2, our statistical performance models have an R^2 of 26.9% and 90.2%. Such high values of the model fit confirms that our performance models can well explain system performance. By looking closely at the models, we can see that the models with our automatically inserted logging statement typically has higher R^2 than the models that are only using web logs. For example, by insert logging statements into two methods in OpenMRS, the fit of the performance model almost doubles (from 26.9% to 46.3%). However, the models that are with inserted logging statements into basic code blocks have a relatively smaller increase of R^2 in comparison to the ones with method-level logging. In the same example of OpenMRS, inserting the logs into basic code blocks only provides 1.6% increase of the R^2.

Our approach does not suggest an overwhelming amount of logging locations for performance modeling. In total, our approach suggests three, two, and four locations for CloudStore, OpenMRS, and ES respectively. We consider such an amount of suggestion as an appropriate amount for practitioners. By measuring the total number of methods in the subject systems, we only suggest to log in less than 0.5% of them. By providing such suggestions to our industrial practitioners, we also received the feedback that such an amount of suggestions is not overwhelming. Hence, practitioners can allocate resource to examine each suggestion and make the final decision of whether to insert logging statements to those locations. Moreover, by manually examining each of the logging locations, we find that **None of the suggested logging locations contain logging statements.** This implies that our approach may provide additional information about the system performance other than what is already known by developers.

> The logging locations suggested by our approach significantly improve the performance models that are with a high model fit. None of those locations initially contain a logging statement.

RQ2: How large is the performance influence by the recommended logging locations?

Motivation.

In the previous research question, we find that, with our approach, we can suggest logging locations that are statistically significant for performance modeling. Even though these logging locations are statistically significant, the effect of the logging location may still be trivial. Therefore, in this research question, we would like to examine the magnitude of the influence on system performance by our suggested logging locations.

Approach.

To understand the magnitude of the influence on system performance by our recommended logging locations, we first calculate Pearson correlation between the system performance, i.e., CPU, and with the appearance of the suggested logging locations. Higher correlation implies that the suggested logging locations may have a higher influence on the system performance.

To quantify the influence, we follow a similar approach used in prior research [31, 35]. To quantify this magnitude, we set all of the metrics in the model (each as a suggested logging location) to their mean value and record the predicted system performance. Then, to measure the effect of every logging location, we keep all of the metrics at their median value, except for the metric whose effect we wish to measure. We double the median value of that metric and recalculate the predicted system performance. We then calculate the percentage of difference caused by doubling the value of that metric. For example, if the CPU is 60% at all metrics with median value and 90% by increasing one log metrics, the effect is 0.5, i.e., $\frac{90\% - 60\%}{60\%}$. The effect of a metric can be positive or negative. A positive effect means that a higher chance of execution the suggested logging location may increase the system performance, e.g., higher CPU usage. This approach permits us to study metrics that are of different scales, in contrast to using odds ratios analysis, which is commonly used in prior research [34].

Results.

The appearance of the suggested logging locations influences the system performance. Table 3 shows that the appearance of the suggested logging locations typically has a strong correlation to system performance. In CloudStore, all of the logging locations have a strong correlation to CPU usage, while the correlations are moderate in OpenMRS. The relative effect shows the influence of one method while controlling all other methods. `DaoImpl.getCurrentSession()` in CloudStore has the largest effect when the appearance of the method is double to its median value: the CPU usage increases 124%. Table 3 shows that even method with a small effect, e.g., `ConceptServiceImpl.getFalseConcept()`, can increase the CPU usage by 19% if doubling its appearance.

The influence on the system performance may be both positive or negative. We find that some suggested logging locations in ES may have a negative influence on the CPU usage of the system, i.e., the higher the appearance of the logging location, the lower the CPU usage. By manually examining those methods, we find that these methods are related to synchronized external dependency, i.e., the invocation of these methods will cause the system to wait, leading to lower CPU usage. By having these logs, developers can consider addressing such synchronized dependency based on how often real-life users call these methods.

Table 2: R^2 values of the statistical performance models built by our approach.

Cloud Store				
Steps:	Web request name	Method name/Block location	R^2 Original	R^2 With logging statement as a metric
Step 1: Web logs only	N/A		90.20%	N/A
Step 2: With method instrumentation	"cloudstore/ "	HomeController.getProductUrl()) (No block)	78.50%	80.50%
	"cloudstore/buy"	DaoImpl.getCurrentSession()(No block)	49.40%	49.50%
	"cloudstore/search"	ItemDaoImpl.findAllByAuthor()	78.00%	81.20%
Step 3: With block instrumentation	"cloudstore/search"	ItemDaoImpl.java, line 233 to 243	81.30%	81.60%

OpenMRS				
Steps:	Web request name	Method name/Block location	R^2 Original	R^2 With logging statement as a metric
Step 1: Web logs only	N/A		26.90%	N/A
Step 2: With method instrumentation	concept/	ConceptServiceImpl.getAllConcepts() ConceptServiceImpl.getFalseConcept()	46.30%	47.80%
Step 3: With block instrumentation	concept/	ConceptServiceImpl.java, line 300 to 302 ConceptServiceImpl.java, line 929 to 930	47.90%	48.00%

ES				
Steps:	Web request name	Method name/Block location	R^2 Original	R^2 With logging statement as a metric
Step 1: Web logs only	N/A		43.80%	N/A
Step 2: With method instrumentation	Web request A	file1.m() file2.n()	75.90%	76.40%
	Web request B	(No significance)	30.00%	N/A
	Web request C	file3.o() file4.p()	42.90%	43%
Step 3: With block instrumentation	Web request A	file1, block.r file2, block.x	70.80%	70.80%
	Web request C	file3, block.y file4, block.z	76.00%	76.30%

"No block" means that the method only has one basic block in the method body.
"No significance" means that none of the methods are significant in the performance model.
We only present the class names, the method names and the file names due to the limit of space, without showing the package names and the full path of the files.

> Our suggested logging locations have influences on system performance; while such influence can be both positive and negative.

RQ3: What are the characteristics of the recommended logging locations?

Motivation.

In the previous research questions, we leverage our approach to suggest logging locations to assist in performance modeling. If we can study the characteristic of these locations in the source code being performance influential, we may provide more general guidance for a developer to log similar locations in the source code.

Furthermore, prior research has proposed various techniques to provide general guidance on logging locations [21, 47] or to monitor hot methods in performance. Our approach may be of less interest if prior techniques also suggest such locations to log.

Approach.

For each of the suggested logging locations, we manually examine the surrounding source code to understand their characteristics. In particular, the size of the source code, such as lines of code, one of the factors prior study used to model logging decisions [47]. Moreover, uncertainty concerning control flow branches is also considered in logging decisions [46]. Therefore, we measure the source lines of code (SLOC) of the suggested methods and blocks and the cyclomatic complexity of the methods that are suggested to be logged.

Furthermore, we massively instrument the execution of all subject systems with JProfiler and Visual Studio Profiling tool [3, 8]. We measure both inclusive and exclusive execution time of each method and rank all the methods by their execution time. We would like to examine whether our suggested methods are one of the hot methods, i.e., with the highest executed time.

Results.

The suggested logging locations are not in complex methods. By measuring the SLOC and cyclomatic complexity, we find that the suggested logging locations are in the methods with small sizes and low complexity. The methods that are suggested to be logged have a SLOC of 4, 5 and 15 in CloudStore, and methods in OpenMRS consists of only 3 and 6 SLOC. In ES, all suggested methods have a SLOC less than 35. Similarly, the values of the cyclomatic complexity of the suggested methods in CloudStore are only 1, 2 and 2; the

Table 3: The influences of our suggested logging locations on system performance.

Cloud Store			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
cloudstore/	HomeController.getProductUrl()	+0.80	+0.19
cloudstore/buy	DaoImpl.getCurrentSession()	+0.70	+1.24
cloudstore/search	ItemDaoImpl.findAllByAuthor()	+0.87	+0.60
cloudstore/search	ItemDaoImpl.java, line 233 to 243	+0.73	+0.25

OpenMRS			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
concept/	ConceptServiceImpl.getFalseConcept()	+0.51	+0.19
concept/	ConceptServiceImpl.getAllConcepts()	+0.53	+0.22
concept/	ConceptServiceImpl.java, line 300 to 302	+0.56	+0.25
concept/	ConceptServiceImpl.java, line 929 to 930	+0.56	+0.22

ES			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
Web request A	file1.m()	-0.27	-0.34
Web request A	file2.n()	+0.81	+1.17
Web request C	file3.o()	-0.40	-0.80
Web request C	file4.p()	+0.56	+0.49
Web request A	file1, block.r	-0.26	-0.39
Web request A	file2, block.x	+0.78	+1.11
Web request C	file3, block.y	-0.11	-0.28
Web request C	file4, block.z	+0.86	+0.88

A relative positive effect means that more appearances of the logging location may result in CPU usage increase.

We only present the class names, the method names and the file names due to the limit of space, without showing the package names and the full path of the files.

same values are merely 1 and 2 in OpenMRS. The small sizes and the low complexity of the methods imply that practitioner may use our approach in tandem with other approaches that are based on source code metrics.

Most of the suggested logging locations are not the performance hotspot. By examining the results of detecting hotspots using both inclusive and exclusive execution time, we find that our suggested logging locations are not typical performance hotspots. In particular, only one of the logging locations (ItemDaoImpl.findAll ByAuthor()) is in the top 10 of hotspots in the source code (excluding methods in the library). We consider the reason is that our approach does not aim to identify the methods that are invoked often, but the ones that can explain the system performance variance. Therefore, our approach may complement the detection of performance hotspots in performance assurances activities.

> The suggested logging locations are typically not in complex methods nor performance hotspots. Performance engineers can use our approach to complement those traditional measurements in performance engineering activities.

5 DISCUSSION

In this section, we discuss the related topics based on our results.

5.1 Performance influence from the inserted logging statement.

The invocation of logging statements themselves has a performance overhead. To minimize such performance overhead, we opt to reduce the instrumentation scope at every run of the system by focusing on only one web request, web page or method at each time. Moreover, we also leveraged async-logging provided by the logging library to reduce overhead. However, introducing those logging statements still brings overhead to the system.

Therefore, we measure the influence of the inserted logging statement to the fit of the model. We consider the invocation to the logging library itself as a method to monitor and create a log metric measuring the times that the logging library is called to generate logs. For every model that we built in our case study (see Table 2), we add the new log metric as an independent variable. By adding this independent variable into the model, we can study whether the log metric provides an increase of R^2, which represents the additional explanatory power of the execution of the inserted logging statement to the system performance. The increase of R^2 measures the explanatory power of the model that is provided only by the execution of the logging statements, but not the software system itself.

The automatically inserted logging statements do not contribute significantly to the performance models. We find that the log metric that measures the execution of the logging statements provides only little explanatory power to the models. In particular, the maximum of the increase of the R^2 is only 3.4% (see Table 2). Therefore, the inserted logging statement do not have a large impact to bias the explanatory power of our suggested logging locations.

5.2 Not all web requests need additional logging.

After applying our approach, inserting logging statements may not provide statistically significantly more explanation power to the model. For example, in the Web Request B of ES, after inserting logging statements into all associated method, none of them are statistically significant in the performance model. Such results imply that over-inserting logging statements into the source code may only provide repetitive information that is already available from other logs, whiling leading to more noise to practitioners [41]. By looking at the web request and the methods that do not need additional logging, we find that these cases are typically simple sequential executions with low complexity. For example, `ItemDaoImpl.findAllByAuthor()` in CloudStore has a loop as an extra basic block. However, our results show that inserting logging statement into the loop would not improve the performance model. That implies that the number of iterations of the loop may not influence performance significantly.

5.3 How long do we need to test performance to suggest logging locations?

Performance testing is a time-consuming task [10]. However, our approach requires multiple iterations of conducting performance tests. Even though it is straightforward to deploy the multiple performance tests in separate testing environments to reduce the time, such solution may still be resource-costly. In order to minimize the cost of the resource, we investigate whether we may shorten the duration of the performance tests and still yield similar results.

For every performance test, we take the data from the period of the first hour, the first two hours and the first three hours. We then follow the same steps as Section 3 and examine whether we can suggest the same locations to insert logging statements. We find that in 4 models, we can achieve the same logging suggestions by only running one hour, two hours and three hours of the test in four, one, and six models, respectively. We need the complete four hours only in two models. This result shows that practitioners may be able to reduce the test duration in practice to receive the suggestion in a more timely manner.

6 RELATED WORK

In this section, we present the prior research related to this paper in three aspects: 1) software performance monitoring, 2) assisting logging decisions and 3) software performance modeling.

6.1 Software performance monitoring

There exist three typical levels of software monitoring techniques. The first, *system monitoring*, monitors the status of a running software based on the performance counters from the system. Examples of such counters include, CPU usage, memory usage and I/O traffic. Rich data from these counters are widely used to monitor system performance [17], allocate system resources and plan capacities [48] or predict system crash [16]. Despite the usefulness of such data, the lack of domain knowledge of the software running on top of the system makes the data difficult to use for improving the system in a detailed level (like improving source code).

The second type of widely used techniques are based on massive *tracing*. The tracing information records every function call that is invoked during the running of the system. Prior research leverage the tracking information to system quality and efficiency [44, 45]. In order to generate such tracing information, tools such as *JProfiler* [3] is widely used in practice and research. The challenge of leveraging such tracing information is the extra overhead from the tracing tools. Such overhead prevent the use of tracing in a large scale system or during the field running of the system, hence tracing is often used in the development environment by developers. Nevertheless, Maplesden et al. took advantage of patterns in tracing information. They built an automated tool to detect such patterns with the goal of improving the performance investigations and the systems' performance [29, 30].

To minimize the overhead from tracing, techniques are proposed to only trace a selected set of function calls, such that the tracing information from the field is possible to be monitored. For example, Application Performance Management tools [9] typically choose REST API call entry points to monitor. However, trace information is often generated automatically without the interference of developers' knowledge. The collected trace information may not all be needed for developers' particular purpose while the actual needed information may be missing.

The third type of monitoring technique is based on logging. Developers write logging statements in the source code to expose valuable information of runtime system behavior. A logging statement, e.g., *logger.info("static string"+ variable)*, typically consists of a log level (e.g., trace/debug/info/warn/error/fatal), a logged event using a static text, and variables that are related to the event context. During system runtime, the invocation of these logging statements would generate logs that are often treated as the most important, sometimes only, source of information for debugging and maintenance of large software systems. The logging information are generated based on developers' knowledge of the system, and are flexible to monitor various information in the code. Due to the extensive value in logs, prior research has proposed to leverage logging data to improve the efficiency and quality of large software systems [14, 15, 24, 33]. The advantage of using logging to monitor and analyze system performance motivates our paper. In particular, with our approach, the prior research that depends on logging may benefit from the extra information that are captured from the suggested logging statements.

6.2 Assist in logging decisions

Although logging is a significant technique for software performance monitoring, the logging practice in general is not as straight forward as one would expect. Logging involves a trade-off between the overhead it can generate and having the appropriate information. In a previous work, Zhao et al. proposed an algorithm that touches such trade-off. They increase the debugging assertiveness by automatically placing logs based on an overhead limit threshold [46]. Even if no overhead existed, there is still a need to balance between too much information and too little information [21].

Aiming to support the logging decisions, many previous works have contributed in ways to understand, automate and suggest opportunities of where to log. Fu et al. performed an empirical study on industry systems categorizing logged snippets of code. Their work also revealed the possibility of predicting where to log according to the extracted logging features [21]. Zhu et al. follow up the work and predict where to log as suggestions to developer. Similarly, a called *Errlog* presented by Yuan et al. indicated the benefits of automatically detecting logging opportunities for failure diagnosis using exception patterns and failure reports [42].

Previous research also presented other aspects to consider when taking logging decisions. Li et al. modeled which log level should be used when adding new logging statements [27]. In a different work, Li et al. studied log changes and modeled those log changes to provide a just-in-time suggestion to developers for changing logs [28]. Different previous research has presented what to log for a diverse set of concerns. Yuan et al. presented *LogEnhancer* that adds causally-related information to existing logging statements. Their focus was on software failures and software diagnosability [43]. Despite of above research effort, there exists no research focus on providing logging suggestions with the goal of monitoring system performance. In contrast with previous research, paper work focus on logging suggestion for performance.

6.3 Performance modeling

Performance modeling is a typical practice in system performance engineering. Due to the more complex nature of performance problems in distributed systems, simple raw metrics might not be enough. Therefore, Cohen et al. introduced the concept and use of *signatures* and *clustering* from logging data and system metrics to detect system states that are of significant impact in the system's performance [17]. With such data, Cohen et al. [16] used TAN (Tree-Augmented Bayesian Networks) models to model the high-level system performance states based on a small subset of metrics without *a priori* knowledge of the system. Brebner et al. have application performance management (APM) data in multiple industry projects to build performance models. However, the models that depend on APM can get very complex, and customization is needed [12] In order to improve the quality of performance modeling and prediction. Stewart et al. [36] consider the inconsistency of usage in enterprise and large e-commerce systems. In their work, they modeled using measurement data and *transaction mix*, and they report a better prediction quality instead of the existing scalar workload volume approach.

Since there could be too many performance metrics to be used in performance modeling, different previous researches address the issue. Xiong et al. [40] propose an automatic creation and selection of multiple models based on different metrics. They execute tests on virtual machines using standard performance benchmarks. Shang et al. [33] presented an approach to automatically group metrics in a smaller number of clusters. They used regression models on injected and real-life scenarios, and their approach outperforms traditional approaches.

Besides the use of regression models, other statistical techniques have been used to facilitate the communication of results, such as control charts [32]. Many different modeling approaches have been summarized by Gao et al. in three categories: rule-based models, data mining models and queueing models. In their work, they used the models to compare the effectiveness of load testing and provide insights on how to better do load testing [22]. Farshchi et al. [18] build correlation model between logs and operation activity's effect on system resources. Such correlation is later leveraged to detect system anomalies.

The rich usage of performance modeling supports our approach that leverages such model to suggest logging locations. We iteratively find the best logging locations that would provide most significant explanatory power to the performance of the system.

7 THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

7.1 External Validity

Our evaluation is conducted on CloudStore, OpenMRS and ES. All subject systems have years of history and there are prior performance engineering research studying these systems' workload [13]. Nevertheless, more case studies on other software in other domains are needed to evaluate our approach. All our subject systems are developed based on either Java or .Net. Our approach may not be directly applicable for other programming languages, especially dynamic languages such as Python. Further work may investigate approaches to minimize the uncertainty in performance characterization of dynamic languages.

Our approach currently only focuses on web application. We leverage web logs in the first step in order to scope down the amount of source code to instrument. However, other researchers and practitioners may adapt our approach by applying our approach by starting on a few hot locations in the source code. Yet, without evaluation with such an approach, we cannot claim the usefulness of our approach on other types of systems.

7.2 Internal validity

Our approach is based on the system performance that is recorded by *Psutil*. The quality of recorded performance can impact the internal validity of our study. Similarly, the frequency of recording system performance by *Psutil* may also impact the results of our approach. Further work may further evaluate our approach by varying such frequency. Our approach depends on building statistical models. Therefore, with a smaller amount of performance data, our approach may not perform well due to the quality of the statistical model. Determining the optimal amount of performance data needed for our approach is in our plan. Although our approach builds statistical models using logs, we do not aim to predict nor

claim causal relationship between the dependent variable and independent variables in the models. The only purpose of building regression models is to capture the relation between logs and system performance.

7.3 Construct validity

Our approach uses linear regression models to model system performance. Although linear regression models have been used in prior research in performance engineering [33, 40], there exist other statistical models that may model system performance more accurately. Our goal is not to accurately predict system performance but rather capture the relationship between logs and the system performance. Further work may investigate the use of other models.

We chose to design our approach in an aggressive manner when deciding potential logging locations. For example, we choose a low p-value to ensure the statistical significance of the logging location. Our approach may miss potential possible logging locations. However, our goal is to prioritize on the precision of the suggestion hence making the suggestion less intrusive to practitioners. By working with our industrial collaboration, we find that a large number of logging suggestions can be overwhelming since practitioners prefer to manually verify each logging location before having actual changes to the source code.

The overhead of the logs may influence system performance. Although we evaluate the impact of logs on system performance by examining the explanatory power of logging statements themselves, the overhead may still impact the results of our approach. Minimizing such overhead is in our further plan.

Our evaluation of our approach is based on modeling system CPU usage. There exist other performance metrics, such as memory and response time, that can be modeled by logs when evaluating our approach. Also, the performance of the subject systems is recorded while running their performance tests. If a logging location is not executed by performance tests, it cannot be identified by our approach. To address this threat, we sought to use the performance test that mimics the field workload from our industrial collaborators. However, a different workload may lead to different performance influencing locations in the source code. Therefore, when applying our approach, practitioners should always be aware of the impact from the workload (the performance tests on the system). Hence, evaluation with more performance metrics and more performance tests may lead to better understanding of the usefulness of our approach.

Although we suggest logging locations for performance assurance activities, we do not claim that they are the only relevant logging locations. Additionally, the R^2 of our models is between 26.9% and 90.2%. The R^2 shows that logs cannot explain all the variance in the system performance. The unexplained variance of performance may due to other performance influencing source code or external influence of the system (e.g., network latency). In our future work, we plan to model other influencing factors of system performance to improve our approach.

Our approach is based on automated code analysis and code manipulation, when changing and rebuilding the software is needed. Such an approach may require extra resources to the performance infrastructure. In our future work, we plan to alter the source code adaptively during the runtime of performance testing or in the field to improve our approach.

8 CONCLUSION

Logging information is one of the most significant sources of data in performance monitoring and modeling. Due to the extensive use of logs, all too often, the success of various performance modeling and analysis techniques often rely on the availability of logs. However, existing empirical studies and automated techniques for logging decisions do not consider the particular need for system performance monitoring. In this paper, we propose an approach to automatically suggest where to insert logging statements with the goal of support performance monitoring for web-based systems. Our approach suggests inserting logging statement into the source code locations that can complement the explanation power of statistical performance models. By evaluating our approach on two open source systems (CloudStore and OpenMRS) and one commercial system (ES), we find that our approach suggests logging locations that improve the statistical performance models and those suggested logging locations have a high influence on system performance while not being traditional complex methods nor performance hotspots. Practitioners can integrate our approach into the release pipeline of their system to have logging suggestions periodically.

ACKNOWLEDGEMENT

We would like to thank ERA Environmental Management Solutions for providing access to the enterprise system used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of ERA Environmental Management Solutions' products.

REFERENCES

[1] 2017. CloudScale Project. (oct 2017). Retrieved Oct 9, 2017 from http://www.cloudscale-project.eu/
[2] 2017. Eclipse Java development tools (JDT). (oct 2017). Retrieved Oct 9, 2017 from http://www.eclipse.org/jdt/
[3] 2017. JProfiler. (oct 2017). Retrieved Oct 9, 2017 from https://www.ej-technologies.com/products/jprofiler/overview.html
[4] 2017. Log4J Async. (oct 2017). Retrieved Oct 9, 2017 from https://logging.apache.org/log4j/2.x/manual/async.html
[5] 2017. .NET Compiler Platform ("Roslyn"). (oct 2017). Retrieved Oct 9, 2017 from https://github.com/dotnet/roslyn
[6] 2017. psutil. (feb 2017). Retrieved Feb 2, 2017 from https://github.com/giampaolo/psutil
[7] 2017. TPC Benchmark W (TPC-W). (oct 2017). Retrieved Oct 9, 2017 from http://www.tpc.org/tpcw/
[8] 2017. Visual Studio Profiling. (oct 2017). Retrieved Oct 9, 2017 from https://docs.microsoft.com/en-us/visualstudio/profiling
[9] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 1–12.
[10] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan. 2016. An Automated Approach for Recommending When to Stop Performance Tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 279–289.
[11] Apache. [n. d.]. Jmeter. http://jmeter.apache.org/. ([n. d.]). Accessed: 2015-06-01.
[12] Paul Charles Brebner. 2016. Automatic Performance Modelling from Application Performance Management (APM) Data: An Experience Report. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16)*. ACM, New York, NY, USA, 55–61.

[13] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-based Database-centric Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 666–677.

[14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1001–1012.

[15] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Transactions on Software Engineering* PP, 99 (2016), 1–1.

[16] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.. In *OSDI*, Vol. 4. 16–16.

[17] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, Indexing, Clustering, and Retrieving System History. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 105–118.

[18] M. Farshchi, J. G. Schneider, I. Weber, and J. Grundy. 2015. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 24–34.

[19] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. 2010. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 32–41.

[20] David A Freedman. 2009. *Statistical models: theory and practice*. cambridge university press.

[21] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 24–33.

[22] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu. 2016. A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 22–32.

[23] FE Harrell. 2001. Regression modeling strategies. 2001. *Nashville: Springer CrossRef Google Scholar* (2001).

[24] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *ICSM '09: 25th IEEE International Conference on Software Maintenance*.

[25] Brian W. Kernighan and Rob Pike. 1999. *The Practice of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[26] Max Kuhn. 2008. Building Predictive Models in R Using the caret Package. *Journal of Statistical Software, Articles* 28, 5 (2008), 1–26.

[27] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which Log Level Should Developers Choose for a New Logging Statement? *Empirical Softw. Engg.* 22, 4 (Aug. 2017), 1684–1716.

[28] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards Just-in-time Suggestions for Log Changes. *Empirical Softw. Engg.* 22, 4 (Aug. 2017), 1831–1865.

[29] David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. 2015. Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 175–186.

[30] David Maplesden, Karl von Randow, Ewan Tempero, John Hosking, and John Grundy. 2015. Performance Analysis Using Subsuming Methods: An Industrial Case Study. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 149–158.

[31] Audris Mockus. 2010. Organizational Volatility and Its Effects on Software Defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 117–126.

[32] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2012. Automated Detection of Performance Regressions Using Statistical Process Control Techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. ACM, New York, NY, USA, 299–310.

[33] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated 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)*. ACM, New York, NY, USA, 15–26.

[34] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. 2010. Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. ACM, New York, NY, USA, Article 4, 10 pages.

[35] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2011. High-impact Defects: A Study of Breakage and Surprise Defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 300–310.

[36] Christopher Stewart, Terence Kelly, and Alex Zhang. 2007. Exploiting nonstationarity for performance prediction. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 31–44.

[37] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. 2017. Continuous validation of performance test workloads. *Automated Software Engineering* 24, 1 (2017), 189–231. https://doi.org/10.1007/s10515-016-0196-8

[38] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. ACM, New York, NY, USA, 247–248.

[39] E.J. Weyuker and F.I. Vokolos. 2000. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering* 26, 12 (Dec 2000), 1147–1156.

[40] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 2013. vPerfGuard: An Automated Model-driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, New York, NY, USA, 271–282.

[41] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 249–265.

[42] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging.. In *OSDI '12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Vol. 12. 293–306.

[43] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proc. of the 16th international conference on Architectural support for programming languages and operating systems*.

[44] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 152–163.

[45] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 12–23.

[46] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. The Game of Twenty Questions: Do You Know Where to Log?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 125–131.

[47] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 415–425.

[48] Zhenyun Zhuang, Haricharan Ramachandra, Cuong Tran, Subbu Subramaniam, Chavdar Botev, Chaoyue Xiong, and Badri Sridharan. 2015. Capacity Planning and Headroom Analysis for Taming Database Replication Latency: Experiences with LinkedIn Internet Traffic. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 39–50.