Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments

Alexander Wert
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
alexander.wert@kit.edu

Jens Happe
SAP Research
Vincenz-Priessnitz-Str. 1
Karlsruhe, Germany
jens.happe@sap.com

Lucia Happe
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
lucia.kapova@kit.edu

Abstract—Performance problems pose a significant risk to software vendors. If left undetected, they can lead to lost customers, increased operational costs, and damaged reputation. Despite all efforts, software engineers cannot fully prevent performance problems being introduced into an application. Detecting and resolving such problems as early as possible with minimal effort is still an open challenge in software performance engineering. In this paper, we present a novel approach for Performance Problem Diagnostics (PPD) that systematically searches for well-known performance problems (also called performance antipatterns) within an application. PPD automatically isolates the problem’s root cause, hence facilitating problem solving. We applied PPD to a well established transactional web e-Commerce benchmark (TPC-W) in two deployment scenarios. PPD automatically identified four performance problems in the benchmark implementation and its deployment environment. By fixing the problems, we increased the maximum throughput of the benchmark from 1800 requests per second to more than 3500.

Index Terms—performance; problem detection; measurement

I. INTRODUCTION

The performance of an application is highly visible to end-users and thus crucial for its success. Response times, throughput and resource consumption affect conversion rates\(^1\), user satisfaction, and operational costs. However, performance problems are usually difficult to detect and even harder to reproduce. Applying systematic approaches (e.g. Software Performance Engineering, SPE [1]) requires specific knowledge and significant expertise. As a consequence, most performance and scalability questions are postponed to the end of the development process (“fix-it-later” approach [1]). Problems introduced early in the development process are identified late and thus expensive to fix as they can be disproportionately disruptive to an application’s implementation and architecture.

As examined by Boehm [2], [3] costs escalate as problems are discovered in later development phases. An example of cost escalation has been given by NASA [4]: Fixing a requirements phase error during design increases repair costs three to eight times. During integration, fault removal becomes 21 to 78 times more expensive. Costs explode by 29 to 1500 times for errors found during operation. Even though these numbers may be different in other contexts, the tendency can be expected to be the same. Therefore, it’s vital to identify performance problems as early as possible.

However, slow performance (low throughput, high response times, or high resource consumption) can have various causes in an application architecture, implementation, or deployment environment. Without sufficient expertise, it is hard to identify the actual cause of a problem. Software engineers need to know typical performance problems that can occur in their application. For each problem, they must know where and how to measure in order to get the necessary data without distorting measurements. In many cases, the necessary performance metrics cannot be collected. These will lead to incomplete and noisy measurement data which in turn make it even harder to draw the right conclusions.

Existing approaches try to identify performance problems based on software architectures [5], [6], load tests [7]–[10] or runtime data [11], [12]. While analyzing the architecture can identify potential problems early, it is limited to a very high level of abstraction. Many causes of performance problems are not reflected in an early architecture and thus will be missed in such an analysis. Load tests take into account all effects of the implementation but are focused on specific scenarios, like identification of resource bottlenecks. Approaches using runtime data to detect and diagnose performance problems can operate with real data but should only be a last resort. Problems are detected way too late to be solved efficiently.

In this paper, we introduce a novel Performance Problem Diagnostics (PPD) that automatically identifies performance problems in an application and diagnoses their root causes. Once software engineers specified a usage profile for their application and setup a test system, PPD can automatically search for known performance problems. Since PPD encapsulates knowledge about typical performance problems (for example, performance antipatterns [13]–[18]), only little per-

\(^1\)Conversion rate is the fraction of visitors of a website who become customers.
To validate PPD, we applied it to an established implementation of the TPC-W industry benchmark [19], a Java-based three-tier enterprise application. We deployed the benchmark in two different test environments. PPD identified four performance problems in the benchmark implementation, the web server, the database, and the infrastructure. Solving these problems increased the maximal throughput of the benchmark from 1800 requests per second to more than 3500.

Overall, we make the following contributions:

1) We introduce a novel approach for performance problem detection and root cause analysis called Performance Problem Diagnostics. PPD systematically searches for known performance problems (cf. [13]–[18]) in three-tier enterprise applications. Once a problem has been found, PPD isolates its root causes as far as possible.

2) We structure a large set of known performance problems [13]–[18] in a novel Performance Problem Hierarchy. To guide PPD’s search, the hierarchy starts from very general problems (or symptoms). Each further level refines the problems down to root causes. The hierarchy allows systematically excluding classes of problems and focusing on the most relevant ones.

3) We define detection strategies for twelve performance problems in the hierarchy. The strategies are based on goal-oriented experiments tailored to trigger a specific problem. Based on the results, heuristics can decide if a problem is assumed to be present and refine the search. For each performance problem, we investigated and compared different heuristics for detecting the problems (see Section III). We chose those heuristics that minimize false positives and false negatives.

4) We evaluated our approach in two steps. First, we determined the detection strategies that are most likely to find a performance problem (see Section III). For this purpose, we evaluated the accuracy of each detection strategy based on ten reference scenarios. Each scenario contains different performance problems which have been injected into a test application.

Second, we evaluated if PPD can detect performance problems in real enterprise applications (see Section IV). PPD successfully identified four performance problems in the TPC-W benchmark, which significantly limited the maximal throughput.

In the following section, we introduce the main concepts of our approach.

II. Automatic Performance Problem Diagnostics

The core idea of our Performance Problem Diagnostics (PPD) is based on the observations that i) particular performance problems share common symptoms and ii) many performance problems described in the literature [13]–[18] are defined by a particular set of root causes. Based on these observations, we create a hierarchical structure of performance problems, their symptoms, and their root causes that simplifies the detection and diagnostics significantly (Section II-A). The hierarchy is based on performance antipatterns known in the literature [13]–[18]. To detect performance problems and diagnose their root cause, we execute a series of systematic experiments that first test for symptoms and then search for more specific performance problems and their root cause (Section II-B). In the following, we introduce the idea of both concepts. A detailed description follows in Section III.

A. Performance Problem Hierarchy

Figure 1 shows an excerpt of the hierarchical structure of performance problems. An extended performance problem hierarchy for a large set of the performance problems known in literature can be found on our website [20]. The hierarchy is structured in categories, symptoms, performance problems, and root causes. The category Occurrences of High Response Times in Figure 1(a) groups common symptoms for the performance problems High Overhead, Varying Response Times, Unbalanced Processing [13], and Dispensable Computations. Symptoms represent the starting point for the performance problem diagnostics. They combine common characteristics of a set of performance problems. Each symptom is refined by more specific performance problems that further limit the set of possible root causes.

![Figure 1. Excerpt of our performance problem hierarchy.](image-url)
example, a request to an online shop takes 10 ms when the store application has been started. After a couple of hours of operation the same request takes more than one second. Such a behaviour can, for example, occur if the application contains Dormant References [21], i.e., the memory consumption of the application is growing over time. The root cause is Specific Data Structures which are growing during operation or which are not properly disposed. Another cause of Varying Response Times is the Traffic Jam performance antipattern. A Traffic Jam occurs if many concurrent threads or processes are waiting for the same shared resources. These can either be passive resources (like semaphores or mutexes) or active resources (like CPU or hard disk). In the first case, we have a typical One Lane Bridge [15] whose critical resource needs to be identified. We focus on Synchronization Points (indicated by semaphores and synchronized methods), Database Locks, and Pools as potential root causes. In the case of limited physical resources, the root cause can only be a specific Bottleneck Resource.

Even though the presented hierarchy is not all-encompassing, it is extensible allowing for integration of further performance problems, symptoms, and root causes. In this process, the definition of new and accurate heuristics is the biggest challenge as discussed in Section V.

To detect performance problems and to identify their root causes, the hierarchy introduced in this section serves as a decision tree that structures the search. Starting from the root nodes (representing symptoms of performance problems), our algorithm looks for more and more specific performance problems and finally root causes. For example, symptoms require top-level metrics such as end-to-end response time or CPU utilization. If a certain symptom has been found, the algorithm systematically investigates its associated performance problems. For each problem (and root cause), we repeat the same process. With each step the problem becomes more specific and requires a more fine-grained instrumentation of the system under test. In the following, we explain the systematic experiments in more detail.

B. Detection Strategies

PPD provides a set of detection strategies which search for performance problems in a System Under Test (SUT), i.e., a deployed application, using goal-driven experiments. Each detection strategy addresses a single performance problem or root cause. It is defined by i) a specific variation of workload characteristics (such as the number of users), ii) a specific instrumentation that allows for goal-driven collection of performance data, and iii) a specific analysis strategy to decide if the problem exists in the system under test.

We use systematic experimentation to observe the effect of changes in the workload on the performance of the system under test. Such dependencies can indicate the existence of performance problems or can confirm a particular root cause. For example, we observe changes of end-to-end response times with respect to the number of users. If the variance of response times increases disproportionately with the number of users, the Varying Response Times are an indication for a Traffic Jam or The Ramp. On a lower level, we can observe the waiting time of threads at a synchronization point. If the waiting times increase significantly with the number of users, the Synchronization Point is a potential root cause for a One Lane Bridge. While the first example decides if potential performance problems exist in the SUT as a whole, the latter identifies the root cause of a performance problem.

All detection strategies are defined once and can then be executed against various applications fully automatically. To achieve this, we define detection strategies so that they can be applied to a class of applications (e.g., Java-based enterprise applications). Each detection strategy encapsulates heuristics for the identification of a particular performance problem. We define the required rules for dynamic instrumentation and workload variation as well as analysis methods (heuristics) that can identify the performance problem using measurement data. Section III gives three examples for detection strategies.

C. Evaluation Method

The automation of PPD and its use of heuristics require a thorough evaluation. We evaluate our approach in two phases. First, we assess the accuracy of detection strategies for each performance problem. A detection strategy is accurate if it can identify a performance problem in various conditions with a minimal number of false positives and false negatives (cf. Section III-B). Second, we evaluate the applicability of our approach to real applications. In particular, we address the following questions:

Q1 Which detection strategies can accurately identify performance problems (with the least number of false positives and false negatives)?
Q2 Can PPD identify performance problems and their root cause in real applications?

Question Q1 addresses the validity of the heuristics and assumptions underlying our detection strategies. Each detection strategy is based on a set of heuristics and requires assumptions about dependencies between application usage and performance metrics. Therefore, detection strategies may be sensitive to specific aspects of applications. For example, if an application contains multiple performance problems, these problems can interact in such a manner that the detection strategies fail. In Section III, we inject different performance problems into a test application. We create ten reference scenarios with different combinations of performance problems and evaluate alternative detection strategies against these scenarios.

Question Q2 addresses PPD’s detection success for real applications which can contain various performance problems and many additional influencing factors. PPD cannot address all of these factors and influences. Evaluating PPD in such a setting helps us to understand its strengths and weaknesses. In Section IV, we apply PPD to the TPC-W benchmark in two different deployment environments.
III. DERIVING HEURISTICS FOR PERFORMANCE PROBLEM DETECTION

Detection strategies are heuristics that need to be carefully chosen. In this section, we refine the concept of detection strategies, introduce a process for evaluating and comparing different strategies, and give three examples. The description of the twelve heuristics used in this paper can be found on our website [20].

For each performance problem, symptom, and root cause, we need a detection strategy that accurately identifies the problem. A detection strategy comprises a workload variation, observed metrics, and an analysis strategy all of which contribute to its accuracy.

- **Workload variation**: Independent workload parameters to be varied from one experiment to the next (e.g., number of users or data size).
- **Observed metrics**: Performance metrics to be collected during each experiment defined by instrumentation rules (e.g., end-to-end response times or waiting times).
- **Analysis strategy**: Analysis of measurement data to decide about the presence of a performance problem.

In order to compare the accuracy of different detection strategies for a certain performance problem, we first define what we understand by **accuracy** in this context. A detection strategy is a heuristic that, based on observed performance data of a system under test, signals if a specific performance problem is present in that system. Based on [22], we define accuracy as a tuple \((1 - r_{fn}, 1 - r_{fp})\), whereby \(r_{fn}\) is the probability that a performance problem is falsely neglected (false negative) and \(r_{fp}\) the probability that a problem is falsely identified (false positive).

In the following, we introduce a set of evaluation scenarios (Section III-A) to quantify the accuracy of competing detection strategies and to answer research question Q1. Based on these scenarios, we can determine the accuracy of each detection strategy (Section III-B). To explain the process, we describe and evaluate two detection strategies for The Ramp (Section III-C and III-D) and one for One Lane Bridge (Section III-E).

A. Evaluation Scenarios

For the definition of reference scenarios, we use a test application in which we inject performance problems (Fault Injection [23]). We use a simple Online Banking system as a representative three-tier enterprise application. A scenario is created by injecting performance problems into the application.

For instance, we inject The Ramp [13] into the application by modifying the SQL queries in a way that complete database tables are retrieved in order to show only a fixed, limited number of entries to the user (Sisyphus Database Retrieval [14]). Executing the scenario leads to increasing response times as database tables grow and results in longer data retrieval requests. Another scenario is created by injecting a One Lane Bridge [15]. In this case, we limit concurrency by synchronizing transactions in a way that only one transaction can be executed at a time. The synchronization point is a software bottleneck limiting the application’s performance.

Injecting different performance problems and root causes leads to a set \(\mathbf{s} = (s_1, \ldots, s_n)\) of different reference scenarios which contain either no, one or a mix of performance problems. The reference scenarios allow evaluating detection strategies with respect to false positives, false negatives and interaction effects between performance problems.

B. Accuracy of Detection Strategies

In order to estimate the error probabilities \(r_{fn}\) and \(r_{fp}\) for a detection strategy we apply the strategy on a predefined set of reference scenarios (introduced in Section III-A) counting the number of false positives and false negatives in the detection result. We define an expectation vector \(\mathbf{v}_p = (v_{p,1}, \ldots, v_{p,n})\) for each performance problem \(p\). \(\mathbf{v}_p\) describes in which scenarios \(s_i\) the problem \(p\) must be detected and in which not:

\[
v_{p,i} = \begin{cases} 1 & \text{scenario } s_i \text{ contains problem } p \\ 0 & \text{otherwise} \end{cases}
\]

Let \(t_p = \{t_{p,1}, \ldots, t_{p,m}\}\) be the set of detection strategies for performance problem \(p\). Applying a detection technique \(t_{p,k}\) on all scenarios \(s_i\) yields a detection vector \(d_{p,k} = (d_{p,k,1}, \ldots, d_{p,k,n})\) describing in which scenarios the considered performance problem has been detected by the applied strategy:

\[
d_{p,k,i} = \begin{cases} 1 & \text{strategy } t_{p,k} \text{ detected } p \text{ in scenario } s_i \\ 0 & \text{otherwise} \end{cases}
\]

The error vector \(e_{p,k} = (e_{p,k,1}, \ldots, e_{p,k,n})\) is the difference \(e_{p,k} = d_{p,k} - \mathbf{v}_p\) and contains the frequencies of false positives and false negatives:

\[
e_{p,k,i} = \begin{cases} -1 & t_{p,k} \text{ falsely neglected } p \text{ in } s_i \\ 1 & t_{p,k} \text{ falsely identified } p \text{ in } s_i \\ 0 & \text{otherwise} \end{cases}
\]

Counting false positives and false negatives in \(e_{p,k}\) and normalizing the sums by the expected number of true positives and true negatives yields the error rates \(r_{fn,p,k}\) for false negatives and \(r_{fp,p,k}\) for false positives, respectively:

\[
r_{fn,p,k} = \frac{(\mathbf{1} - \mathbf{v}_p) \cdot e_{p,k}}{\mathbf{1} \cdot (\mathbf{1} - \mathbf{v}_p)} \
r_{fp,p,k} = \frac{(\mathbf{1} - \mathbf{v}_p) \cdot e_{p,k}}{\mathbf{1} \cdot (\mathbf{1} - \mathbf{v}_p)}
\]

We use the error rates as an accuracy metric for comparing different detection strategies. Considering a performance problem \(p\), a detection strategy \(t_{p,1}\) with error rates \((r_{fn,p,1}, r_{fp,p,1})\) is more accurate than a detection strategy \(t_{p,2}\) with error rate \((r_{fn,p,2}, r_{fp,p,2})\), if

\[
\begin{cases} r_{fn,p,1} < r_{fn,p,2} & \text{if } r_{fn,p,1} \neq r_{fn,p,2} \\ r_{fp,p,1} < r_{fp,p,2} & \text{if } r_{fn,p,1} = r_{fn,p,2} \end{cases}
\]

As neglecting a performance problem is worse than falsely identifying one, Equation (5) implies that minimizing the rate...
for false negatives has a higher priority than minimizing false positives.

In the following, we apply the accuracy metric defined above for the identification of a good detection strategy for The Ramp [13]. The Ramp describes the problem of growing response times over the operation time of an application. We introduce two detection strategies (called “Direct Growth” and “Time Windows”) to discover this tendency. Both strategies perform differently in our reference scenarios illustrating the process of deriving good detection strategies.

C. Direct Growth (DG)

The “Direct Growth” (DG) detection strategy identifies growing response times over the measurement time of a system under test. It compares response times measured in the beginning to response times measured at the end. If the comparison yields a significant difference, we assume that The Ramp is present. In the following, we describe the experiment setup, the analysis of results and the evaluation of this strategy. We also discuss the weaknesses of this strategy, which ultimately lead to the introduction of the “Time Windows” detection strategy.

Experiment Setup: To trigger The Ramp, the load driver executes the usage profile defined by software engineers against the SUT. The detection strategy requires only one experiment with predefined duration $D$. During the experiment, the load driver submits a fixed workload intensity $w$ to the SUT, while end-to-end response times are observed. Additionally, for each measured response time an observation time stamp is captured. The result of such an experiment is an ordered series $R_t = (t_1, \ldots, t_n)$ of response times with corresponding time stamps $T = (t_1, \ldots, t_n)$ where $t_i$ is the time stamp of $r_i$ for all $1 \leq i \leq n$.

Analysis: In order to decide if response times increase over time, the detection strategy divides the response time series $R_t$ into two subsets $R_1 = (r_1, \ldots, r_k)$ and $R_2 = (r_{k+1}, \ldots, r_n)$. The two subsets span approximately equal time intervals so that $t_{k-1} - t_k \approx t_n - t_{k+1}$.

The “Direct Growth” strategy detects The Ramp, if the response times $R_2$ are significantly larger than $R_1$ indicating growing response times over time. For this purpose, we apply a t-test on both subsets. Let $R_i$ be the random variable for subset $R_i$, $E[R_i]$ its expected value and $\bar{R}_i$ the mean value of $R_i$. The following null hypothesis applies for the t-test: $H_0 : E[R_1] = E[R_2]$. If the t-test rejects $H_0$ and $\bar{R}_1 < \bar{R}_2$, the response times $R_2$ are significantly larger than $R_1$ and we assume The Ramp to be present in the SUT.

Evaluation: We evaluated the DG strategy using three different workload intensities $w_1, w_{10}$ and $w_{50}$ (1, 10 and 50 concurrent users). Table I shows the detection results. The ten columns in the middle represent ten reference scenarios as defined in Section III-A. The first row contains the expectation vector $v_{\text{ramp}}$. The Ramp occurs in the scenarios 1, 2 and 10. The following three rows contain the detection vectors for the DG strategy for three different workload intensities. The last two columns contain the error rates for false negatives $r_{fn}$, false positives $r_{fp}$ respectively.

For the DG strategy, the error rates are high, independent of the workload intensity $w$. With a low workload ($w_1$), The Ramp cannot be identified as the response times grow too slow in the scenarios 1, 2 and 10. Using a high workload ($w_{50}$) leads to many false positives. Scenarios 6 and 8 contain software bottlenecks which lead to growing response times, if the workload is permanently high. In these scenarios, the DG strategy falsely identifies The Ramp since the bottlenecks lead to similar behavior. The detection results are not satisfactory in any case. Therefore, we investigate an alternative detection strategy for The Ramp based on time windows.

D. Time Windows (TW)

The “Time Windows” (TW) detection strategy is based on the observations that i) high workload intensities push The Ramp behaviour faster than low workload intensities and ii) bottleneck effects have to be excluded. The TW strategy addresses both conflicting requirements as described in the following.

Experiment Setup: To deal with the conflicting requirements, we divide each experiment into two phases: A stimulation phase and an observation phase. During the stimulation phase, the TW strategy pushes a potential The Ramp antipattern by submitting a high workload to the SUT. In this phase, no measurements are taken. During the observation phase, the TW strategy applies a closed workload with only one user and a short think time. This workload guarantees that requests are not processed concurrently, allowing us to exclude synchronization problems. In this phase, we capture a fixed number of end-to-end response times.

In order to observe the response time progression during operation, we repeat this experiment increasing the duration of the stimulation phase. In this way, we get $n$ chronologically sorted sets $R_i$ (time windows) each containing a fixed number of response time measurements. Figure 2 shows the mean response times of scenario 1 and 3 after different stimulation phases. Scenario 1 contains The Ramp while scenario 3 does not.

Analysis: We compare the response times of the SUT for different stimulation times to detect increases in response time during operation. For this purpose, we perform pairwise t-tests on neighbouring time windows.

Two sets $R_i$ and $R_{i+1}$ are neighbouring if $R_{i+1}$ is the time window with the next longer stimulation time compared to $R_i$. For $R_1, \ldots, R_n$, we perform $n-1$ t-tests $T_i$ with null
hypothesis \( H_0 : E[R_i] = E[R_{i+1}] \). If t-test \( T_i \) rejects \( H_0 \) and \( R_i < R_{i+1}, \) \( R_{i+1} \) is significantly larger than \( R_i \). If this applies for all t-tests, we consider that response times grow significantly with the operation time and, thus, assume The Ramp to be present in the SUT.

**Evaluation:** We applied the TW detection strategy on the same ten scenarios as before. For the TW strategy, both error rates are zero, implying that the TW strategy is more accurate than the DG strategy for The Ramp as it overcomes the disadvantages of the DG strategy. It triggers a potential The Ramp behaviour during its stimulation phase. Measurements are only taken during the observation phase, which avoids any concurrency or synchronization problems. Based on these results, we use the “Time Windows” strategy to detect the antipattern The Ramp in a system under test. In the following, we introduce the detection strategy for One Lane Bridge.

**E. Detection Strategy for “One Lane Bridge”**

A One Lane Bridge (OLB) [15] occurs, if a passive resource limits the concurrency in an application. Passive resources can be for instance mutexes, connection pools, or database locks. In the following, we introduce the detection strategy for the One Lane Bridge antipattern selected due to its low error rate with respect to the reference scenarios.

**Experiment Setup:** Since a One Lane Bridge is a typical scalability problem, we are interested in the performance behaviour with respect to an increasing level of concurrency. To detect this antipattern, we define a series of experiments observing the end-to-end response time while increasing the number of users for each experiment. The strategy increases the number of users until i) a resource is fully utilized (i.e., its utilisation is larger than 90%), ii) response times increase more than 10 times, or iii) the maximum number of potential concurrent users is reached. The experiments yield \( n \) sets of response times \( R_1, \ldots, R_n \) where \( n \) is the number of experiments and \( i + 1 \) is the experiment with the next higher number of users compared to experiment \( i \) \((1 < i < n)\).

In order to distinguish an OLB from a Bottleneck Resource, we additionally measure resource utilization during each experiment. Figure 3 illustrates the measurement results for three different scenarios: i) the SUT contains an OLB, ii) the SUT contains no problem, and iii) the SUT contains a bottleneck resource (BR). The graphs on the left hand side show the average response times with respect to the number of users. The graphs on the right hand side show the corresponding mean CPU utilization.

If the SUT contains an OLB, its critical passive resource leads to strongly increasing response times for an increasing number of users. Additionally, CPU utilization is low since the throughput is limited by the passive resource. If the CPU is a Bottleneck Resource (BR), response times increase and CPU utilization is high. Thus, we do not assume an OLB to be present. Finally, if no performance problem occurs, response times and CPU utilization increase only moderately.

**Analysis:** The analysis is based on the three cases described above. It identifies the number of users for which response times increase significantly and checks the utilisation of resources for all subsequent experiments. Strongly increasing response times and low resource utilisation are indicators for an OLB.

Let \( n \) be the number of experiments and \( u_1, \ldots, u_n \) the number of users for each experiment. The number of users increases for each experiment, so that \( u_i < u_{i+1} \) for all \( 1 \leq i < n \). To detect if there is an experiment \( j \) \( \in \{1, \ldots, n\} \) so that response times increase for all subsequent experiments, the analysis performs \( n-1 \) pairwise t-tests \( T_j \) on neighbouring sets \( (R_i, R_{i+1}) \) of response times and compares their mean values (cf. Section III-D). If a \( j \) exists so that all t-tests are rejected for \( R_j \) to \( R_n \), we assume that response times increase significantly for each experiment following \( j \).

If additionally the mean utilization of all resources is low (smaller than 90%) for all subsequent experiments \( (i \geq j) \), we assume that the SUT contains an OLB. Since none of the hardware resources causes the longer response times (due to their low utilization), the only other reasonable explanation is a software resource that limits the concurrency in the application. Otherwise, if the utilization of at least one resource exceeds the threshold (90%), we cannot conclude that the SUT contains an OLB. In this case, the analysis identifies the resource with a utilisation of more than 90% as a Bottleneck Resource.

**Evaluation:** The results demonstrate that the detection strategy can successfully identify bottleneck resources in all reference scenarios.

So far, we evaluated individual detection strategies based on a set of reference scenarios with injected performance problems. We chose the detection strategies that achieved the lowest error rates for these scenarios. In such a controlled setup, our approach works well. However, to understand if...
PPD can find performance problems outside a controlled environment (and answer research question Q2), we apply it to a real enterprise application in the following section.

IV. SEARCHING FOR PERFORMANCE PROBLEMS IN THE TPC-W BENCHMARK

We use the TPC-W Benchmark [24] for evaluation of our approach. TPC-W is an official benchmark to measure the performance of web servers and databases. Thus, we expect it to be tailored for high performance. Finding performance problems there (if any) is especially challenging (and interesting). In the following, we explain the system under test (Section IV-A), describe how PPD helped us to identify and solve four performance problems (Section IV-B), and explain how PPD identified one of these problems as an example (Section IV-C).

A. System under Test

The TPC-W Benchmark [24] emulates an online bookstore providing twelve different request types for browsing and ordering products and two request types for administrative purposes. The emulated bookstore meets the requirements of a realistic enterprise application, which is essential for the evaluation of our approach.

![Fig. 4. Experimental Setup](image)

Figure 4 shows the architecture and the two different setups for evaluation. Both setups comprise three nodes which are connected with a 100 MBit/s Ethernet in Setup A and a 1 GBit/s Ethernet in Setup B. We use a Java-Servlet implementation of TPC-W [19] which is deployed on an Apache Tomcat 6 Web Server. The bookstore accesses a MySQL 5.0.95 database on a separate machine through JDBC. A dedicated Measurement Control Node executes the performance problem diagnostics. The PPD component encapsulates our algorithm implementing detection strategies for known performance problems. Lightweight Satellites collect measurement data from the SUT’s nodes. The satellites automatically instrument the application using Javassist [25]. The instrumentation is based on rules for pattern-matching and is thus independent of the actual application. For example, one instrumentation rule states “Measure the response time of all calls to JDBC”. The PPD component analyses the data as defined by its detection strategies. Software engineers specify the SUT’s Usage Profile which encapsulates typical user behaviour. For this case study, we use a fixed sequence of TPC-W requests.

B. Results

Altogether, PPD identified four performance problems in the benchmark and its setup. The problems were located in the benchmark itself, the web server, the database, and the infrastructure. By resolving all problems, we increased the throughput of the benchmark form 1800 req/s to more than 3500 req/s. In the following, we describe the results of PPD as well as the performance problems in more detail.

First, PPD identified a OneLane Bridge (OLB) in the TPC-W application deployed in Setup A. A pool of database connections limits the concurrency within the application server. It inhibits the full usage of all available resources. As a consequence the maximal throughput of the benchmark does not exceed 1800 req/s (Curve 1 in Figure 5(a)). The maximum utilisation in this setup is approximately 75% for the CPU of the web server’s CPU and 82% for the network connection (cf. Table II).

We could solve this problem by increasing the bandwidth of the Ethernet connection as a Resource Bottleneck (99% network utilisation). The additional traffic leads to collisions that even result in a declining throughput for high load (Curve 2 in Figure 5(a)) and long response times (Bar 2 in Figure 5(b)). In order to exclude network performance

---

**TABLE II**

<table>
<thead>
<tr>
<th>Users</th>
<th>CPU Utilization</th>
<th>Network Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>6.5</td>
<td>7.1</td>
</tr>
<tr>
<td>10</td>
<td>33.6</td>
<td>35.2</td>
</tr>
<tr>
<td>20</td>
<td>56.1</td>
<td>51.1</td>
</tr>
<tr>
<td>30</td>
<td>65.8</td>
<td>74.3</td>
</tr>
<tr>
<td>40</td>
<td>75.5</td>
<td>79.7</td>
</tr>
<tr>
<td>50</td>
<td>71.2</td>
<td>81.2</td>
</tr>
</tbody>
</table>

---

**Fig. 5. Evaluation on TPC-W**

(a) Throughput

(b) Response Times for complete usage profile for 50 users
problems, we redeployed the TPC-W benchmark to Setup B which provides a higher network bandwidth. In Setup B, the maximal throughput of the benchmark increased to 2200 req/s (Curve 3 in Figure 5(a)).

Despite the increased performance, TPC-W cannot fully utilize the web server’s computational resources (cf. Column 3 in Table II). Running PPD against the benchmark in Setup B, yielded the database connection pool as an OLB, analogously to its first execution. In this case, an increased pool size could not solve the problem. The connection pool remained a software bottleneck. Therefore, we exchanged the default implementation of the Apache Tomcat connection pool by an alternative implementation tailored for high throughput (BoneCP [26]). This step resolved the One Lane Bridge and increased the maximal throughput to 2700 req/s (Curve 4 in Figure 5(a)).

Next, PPD identified a performance problem in the database. The CPU utilisation of the web server still does not exceed 80%, but its throughput is limited and response times increase for a larger number of users. PPD detects that the response times of database calls grow disproportionately, but no resource (including the CPU, network, and disk of the database server) is utilized to capacity. Therefore, PPD identifies the database as a One Lane Bridge, but cannot further pinpoint its root cause.

Since no physical resource is limiting performance, there must be software resources in the database (e.g., table locks) causing the problem. The default storage engine of MySQL 5.0 (MyISAM) only supports table locking as internal locking strategy. If the engine has to process many transactions in parallel, this can lead to unnecessary contention. Changing the storage engine of the MySQL server to InnoDB (which supports row locking) further increased the maximal throughput to 3500 req/s and the CPU utilisation of the web server to 100%.

With the last improvement, we were able to solve all performance problems such that the computing resources of the web server are fully utilized. While solving the performance problems had to be done manually, the identification and root cause diagnosis is fully automated. Executing our diagnostics approach with the hierarchy depicted in Figure 6 took about three hours including the execution of experiments and analyses of data. In the following, we illustrate the diagnostics process for the first case, the diagnostics of the database connection pool as a One Lane Bridge.

C. Performance Problem Diagnostics in Action

Figure 6 shows the results for the decision tree for the initial setup of the benchmark. PPD identifies Varying Response Times in the benchmark implementation. A further investigation excludes The Ramp and identifies a potential Traffic Jam in the system. Therefore, PPD does not investigate more specific problems of The Ramp such as Dormant References but focuses on problems causing a Traffic Jam. By applying the strategy described in Section III-E, PPD can identify a One Lane Bridge as the cause for the Varying Response Times. In the following, we describe how PPD diagnoses the root cause of the One Lane Bridge in more detail.

For each individual step of the diagnostics, PPD automatically instruments specific parts of the application code and evaluates the resulting measurement data. To pinpoint the root cause of the One Lane Bridge, PPD captures the response times of methods that can be linked to a specific root cause, like database calls for Database Locks. In order to identify such methods, PPD inspects the implemented interfaces of all Java-Classes found in the Class-Path and compares their methods to the method signatures searched for. It uses Javassist [25] to dynamically insert code-snippets for response time measurements at runtime. For example, all classes implementing JDBC are instrumented in order to decide if Database Locks cause the problem. When the instrumentation is in place, the workload of the application is varied according to our heuristics. If the response time of the JDBC-Calls increases disproportionally with the workload while no physical resource is fully utilized, DB locks are a potential root cause of the observed performance problem. For the TPC-W benchmark, PPD excludes Synchronized Methods and Database Locks as potential root causes in its first iteration. Instead, it identifies the database connection pool as limiting factor for application performance. The operation for retrieving database connections from the connection pool exhibits a similar increase in response times as observed for the overall response times. Therefore, in this case the connection pool to the database appears to be the potential root cause for the One Lane Bridge.

D. Summary

PPD identified four performance problems and isolated their root cause by systematically narrowing down the search space. In the initial setup, the size of the database connection pool, its default implementation, the network bandwidth, and the storage engine of the database limit the maximal throughput to 1800 req/s. Solving these problems increased TPC-W’s maximal throughput to more than 3500 req/s.

Based on these promising results, we can state that our approach can diagnose performance problems in real applications and detect their root cause (answering question Q2). However, the threats to validity of the results and the assumptions and limitations of our approach require further discussion.
V. Assumptions & Limitations

In the following, we discuss the main assumptions and limitations of our approach: The choice of representative usage profiles, the nature of performance problems, PPD’s general applicability (threats to validity), and the usage of heuristics.

Availability of Usage Profiles: In order to execute experiments, PPD requires a usage profile generating load on the SUT. A usage profile describes the typical behaviour of the application users. The definition of typical usage profiles (or workloads) is a well-known part of load testing using tools like LoadRunner [27]. The effort to define a usage profile can vary depending on the complexity of the application and the required tests. The quality of the usage profile can be critical for the results. While a well-chosen usage profile can trigger rare performance problems, a badly-chosen one may hide even simple problems [8]. In this paper, we assume that software engineers provide a representative usage profile for their application. To provide guidance and simplify the task for software engineers, PPD can be combined with approaches for usage profile optimization (such as [8]).

The Nature of Performance Problems: The performance problems analysed in this paper can be tracked to a specific part in the source code or a specific resource. However, this may hold for all performance problems. Some are the result of the sheer size and complexity of an application. They are distributed over various places in the source code. In such cases, PPD may be only able to detect the problem, but cannot isolate the root causes.

Threats to Validity: The detection strategies presented in Section III have been developed for typical three tier enterprise applications. They apply heuristics based on observations for their application. To provide guidance and simplify the task for software engineers, PPD can be combined with approaches for usage profile optimization (such as [8]).

VI. Related Work

The work in the field of performance problem detection can be categorized along design, implementation & test, and operation within the software lifecycle.

Existing model-based approaches for performance problem detection employ performance models [28] or annotated architectural models [5], [6] to identify performance problems during the design phase. Trubiani et al. [6] predict the performance based on architectural models which are annotated with performance-relevant characteristics. In order to identify performance problems, the authors analyse the resulting predictions with predefined detection rules for individual performance problems. Such model-based approaches can be applied early during design. However, their granularity of detection is limited to a very high level of abstraction. In particular, architectural models do not capture implementation details which often are the source of performance problems. Thus, with model-based approaches alone, many performance problems remain undetected.

Approaches dealing with performance problem detection during implementation & testing can overcome this disadvantage, as they work with implementation artifacts. Most approaches targeting this phase apply performance testing techniques in order to find proper test cases and input data [8]–[10], [29], [30]), identify performance regressions [31], [32] or detect performance bottlenecks [7], [8]. Grechanik et al. [8] introduce an approach for systematic selection of input data for performance tests, so that performance problems can be detected more effectively. Furthermore, the authors describe an approach for automatically identifying performance bottlenecks. For this purpose, they analyse methods along different test cases and their contribution to resource consumption. Their approach is able to find a set of potential bottleneck methods in an application. However, the work of Grechanik et al. is limited to one particular performance problem. We introduce a general concept for identifying different kinds of performance problems based on a combination of systematic search and goal-oriented experimentation.

In order to identify performance problems during operation, existing approaches [11], [12], [33]–[36] apply monitoring techniques to gather performance measurement data. Parsons et al. [11] reconstruct a run-time design model of a component-based software system by analysing the collected data. The design model is then examined with respect to performance antipatterns. As monitoring actions affect the performance of the software under observation, monitoring should be used cautiously during operation. Therefore, some approaches [12], [35], [36] apply dynamic instrumentation for data collection, which maintains the ability to gather all required data. However, all approaches applied during operation share the major drawback that their diagnoses come way to late in order to solve performance problems efficiently.

Miller et al. [12] introduce the idea of dynamically instrumenting the system under test for a goal-oriented search for performance problems, closely related to our work. They report on a tool named Paradyn which is designed for performance evaluation of parallel and distributed software systems. Paradyn encapsulates two essential concepts. First, Miller et al. realize dynamic instrumentation for flexible data collection. Second, a hierarchical search model narrows down performance problems to their actual place and time of occurrence. In our work, we take up the idea of Miller et al. and extend it by an advanced experimentation approach which makes the search for performance problems even more effective.
In this paper, we presented Performance Problem Diagnostics (PPD), a novel, automated approach for performance problem detection and root cause analysis. PPD significantly simplifies the performance validation of enterprise applications. We combined systematic search based on a decision tree with goal-oriented experimentation. For this purpose, we structured performance problems known in the literature in a Performance Problem Hierarchy, which guides the search. For twelve of these performance problems, we developed detection strategies based on heuristics. We evaluated each detection strategy with ten different test scenarios and chose those that proved to be most accurate.

In addition to the evaluation of individual detection strategies, we applied PPD to a 3rd party implementation of the well established TPC-W benchmark. We deployed the benchmark in two environments. PPD identified four performance problems in the benchmark implementation, the web server, the database, and the infrastructure. By fixing these problems, we were able to increase the maximum throughput of the benchmark from 1800 requests per second to more than 3500. As such, the performance problems had a significant effect on the benchmark results.

PPD allows software engineers to automatically search for performance problems in an application with relatively low effort. Lowering the burden of performance validation enables more regular and more sophisticated analyses. Performance validation can be executed early and on a regular basis, for example, in combination with continuous integration tests.

Based on the encouraging results presented in this paper, we plan to integrate our approach with the development infrastructure at SAP. This will allow us to stepwise improve and refine our approach and extend the range of performance problems that can be identified.

ACKNOWLEDGEMENTS

We would like to thank Dennis Westermann, Roozbeh Farahbod, and Christoph Heger for their fruitful comments and constructive feedback. This work was partially funded by the German Research Foundation (DFG) grant RE 1674/6-1.

REFERENCES