Expert-Guided Automatic Diagnosis of Performance Problems in Enterprise Applications

Christoph Heger*, André van Hoorn†, Dušan Okanović†, Stefan Siegl*, Alexander Wert*
*NovaTec Consulting GmbH, Competence Area APM, D-70771 Leinfelden-Echterdingen, Germany
{christoph.heger, stefan.siegl, alexander.wert}@novatec-gmbh.de
†University of Stuttgart, Institute of Software Technology, D-70569 Stuttgart, Germany
{van.hoorn, dusan.okanovic}@informatik.uni-stuttgart.de

***Abstract*—Application performance management (APM) is a necessity to detect and solve performance problems during operation of enterprise applications. While existing tools provide alerting and visualization capabilities when performance requirements are violated during operation, the isolation and diagnosis of the problem's real root cause is the responsibility of the rare performance expert, often resulting in a boring and recurring task. Main challenges for APM adoption in practice include that initial setup and maintenance of APM, and particularly the diagnosis of performance problems are error-prone, costly, and require a high manual effort and expertise. In this paper, we present preliminary work on *diagnoseIT*, an approach that utilizes formalized APM expert knowledge to automate the aforementioned recurring APM activities.**

I. Introduction

Various model-based and measurement-based techniques exist to evaluate the performance of software systems in all lifecycle phases [3], [5]. Despite of their major business impact, performance problems during operation are still omnipresent in business-critical enterprise applications (EAs) [9]. Examples of such problems range from response time violations up to complete service unavailability [12]. Common reasons for these problems are architectural or aging-related issues [4], [17], [18].

There is a rapidly growing market of powerful commercial and open-source application performance management (APM) tools that support the collection and visualization of performance-relevant measures covering the entire EA stack [11], [13], [20]. However, APM practice still requires enormous manual effort and expertise, particularly for setting up and maintaining APM configurations (e.g., deciding which software methods to instrument) as well as diagnosing the root causes of performance problems (e.g., an N+1 problem in the database access [18]). These manual tasks are error-prone, costly, and frustrating for the involved performance experts because various tasks and patterns are recurring. Particularly regarding the root cause analysis of performance problems, today's tools give little or no support. On the other hand, researchers have proposed approaches to detect performance problems using model-based and measurement-based techniques [8], [15], [19], [21]—many of them being based on well-known performance anti-patterns [17], [18]. Most of them focus on design and test time rather than analysis of performance information from production. Also, their main focus is on the architectural level rather than concrete code level.

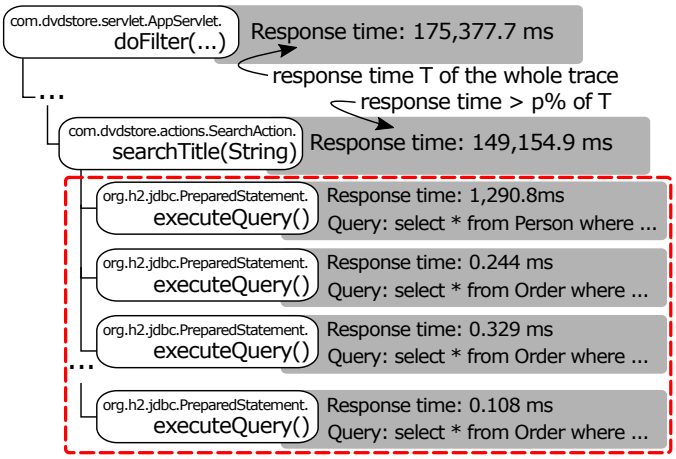

Fig. 1. Example trace including an N+1 problem (in red rectangle)

In this paper, we present the preliminary work on our *diagnoseIT* approach, which aims to address the aforementioned challenges. The core idea is to formalize APM expert knowledge and to use it to automatically execute recurring APM tasks such as the configuration of a meaningful EA instrumentation using APM tools and to diagnose performance problems to isolate their root cause. *diagnoseIT* is designed to be independent of specific APM solutions.

The remainder of this paper is organized as follows: Section II emphasizes the addressed problem and states our vision. Section III outlines our *diagnoseIT* approach—focusing on the trace-based diagnosis of performance problems. Section IV discusses related work. Finally, Section V draws the conclusions and outlines future work.

II. Problem Statement and Vision

In order to emphasize the problem that we address with our *diagnoseIT* approach, we consider a typical process performed by an APM expert in this section.

State-of-the-art APM tools provide a detailed white-box view into EA system stacks—ranging from system-level monitoring (e.g., CPU, memory and network utilization) up to detailed execution traces also spanning multiple nodes includ-

In Proceedings of the 12th European Dependable Computing Conference (EDCC '16)

ing the client devices in distributed EAs. Figure 1 depicts an excerpt of a real trace from an example system. Analysis of the given trace reveals an N+1 problem in the application: one database query with a larger result set is followed by a sequence of short database queries to items obtained in the result set of the initial query [18]. The induced communication overhead could be avoided using a query with join operator, to fetch all the data at once. Common APM tools represent the execution traces with software method caller/callee relationships as call trees. Each method execution is augmented by additional runtime information such as the measured response time, exclusive time,[1] and CPU time.

As previously mentioned, performance problems in EAs are common. Usually, these problems manifest themselves by symptoms such as increased service response times. It is the task of the APM expert to diagnose such a problem, i.e., to identify its respective root cause(s). It can be assumed that a trace as depicted in Figure 1 is or can be collected for each request to a system-provided service. The common procedure is to manually inspect execution traces in a representation as depicted in Figure 1. It needs to be emphasized that this may include hundreds or thousands of traces. It can be easily seen that this task is extremely time-consuming and error-prone. Moreover, APM experts tend to see the same problems (anti-patterns), such as the one in Figure 1, over and over again, because they represent the common pitfalls that happen often in practice.

Before receiving initial execution traces, the APM expert has already made major decisions regarding the level of detail visible in execution traces. The main goal of the initial configuration is to find the right balance between the monitoring overhead and the quality of the obtained data. If the quality of the obtained data is high, it becomes easier to perform analysis and locate performance problems. However, this comes with a price—the system perturbation, e.g., in terms of monitoring overhead, rises. This overhead is inevitably introduced by the APM tool, but is consciously accepted as far as it provides meaningful information for later analysis. The problem is that the initial configuration usually requires a large number of parameters that are used to define the behavior of the monitoring tool. Even if the analysis is performed by an expert who has the required experience in performance problem detection and root cause analysis, it can take several tries to figure out the right configuration that will provide adequate results, while inducing the minimal possible amount of overhead. In case the problem cannot be diagnosed based on the current information, the APM tool needs to be instructed to refine the instrumentation, e.g., to get a higher resolution of caller/callee relationships. In case of very large systems, the diagnosis procedure becomes an even more daunting task. Furthermore, analysis results for one trace cannot be re-used for the next one, and every problem must be treated as unique, although it might be similar to some previous ones.

[1] A method's exclusive time is the difference between its response time and the total response times of the methods invoked by it.

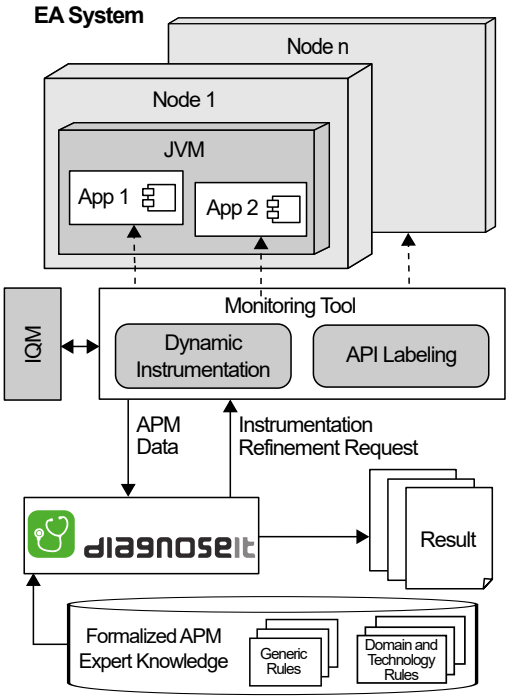

Fig. 2. Position of *diagnoseIT* in monitoring

The core idea of our *diagnoseIT* approach is to automate various of the current manual but recurring APM tasks based on an extensible repository of APM expert knowledge. This APM expert knowledge includes best practices for a meaningful instrumentation of EAs and, particularly, strategies to efficiently diagnose common and recurring performance problems. Being connected to an APM tool, *diagnoseIT* automatically analyzes incoming execution traces and instructs the APM tool to refine the instrumentation to ask for additional information if needed. At the same time, *diagnoseIT* makes sure that at every point in time a meaningful tradeoff between data quality and system perturbation is satisfied. Depending on the current data quality, *diagnoseIT* may give high-level results of root causes such as the slowest method in the trace or very detailed natural language results such as an object-relational (O/R) mapping configuration error in a specific technology (e.g., Hibernate). Results that are reported comprise two aspects: *a.)* qualitative information and *b.)* quantitative information. Qualitative information contains information like a problem's location, type, anti-pattern, and details on its manifestation, while quantitative information contains the impact of the problem, e.g., response and execution times, counts of calls, etc.

The core elements of our approach are that *diagnoseIT* *i.)* automates the steps of diagnosing common and recurring performance problems based on an extensible set of APM expert knowledge, *ii.)* provides a human-understandable report of the root cause, *iii.)* manages a meaningful instrumentation of the EA, and *iv.)* is independent of any specific APM tool.

III. DIAGNOSEIT APPROACH

A high-level view on the *diagnoseIT* approach is depicted in Figure 2. *diagnoseIT* receives performance measurements from APM tools, including detailed execution traces. In case a performance problem is detected, based on the given information, *diagnoseIT* employs strategies from formalized APM expert knowledge to diagnose the problem and reports results. If additional information is needed, a request is sent to the respective APM tool. A dedicated component (Instrumentation Quality Manager—IQM) takes care of a reasonable trade-off between the requested/provided information detail and the system perturbation (including overhead).

It is expected that the underlying APM tools support *i.)* dynamic instrumentation and *ii.)* API labeling. Dynamic instrumentation allows changes to the monitoring configurations (including instrumentation) at runtime without the need to restart the system. API labeling is used for identifying the nature of performance problems, for example, which API is used—e.g., Hibernate, JDBC. If any of these features is not available, *diagnoseIT* will work with the data that is available. It needs to be emphasized that *diagnoseIT* works on any quality of the provided data. However, the higher the quality, the more detailed the diagnosis results will be.

The diagnosis of a problematic trace is performed using an extensible set of rules contained in the formalized APM expert knowledge (Figure 2). Rules are organized hierarchically and can be grouped into *i.)* generic and *ii.)* domain/technology-specific rules. Generic rules are used to narrow down the problem location up to a level which is independent from a specific domain or technology. Domain- and technology-specific rules provide more details, depending on the underlying system's specifics. Figure 3 illustrates the rule-based diagnosis process based on the example from Section II (Figure 1).

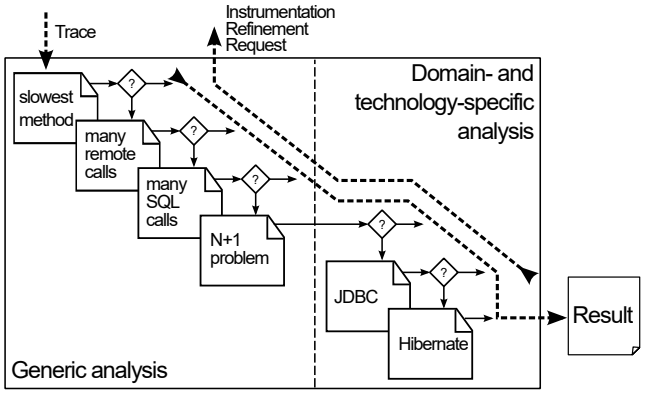

Fig. 3. Exemplary rule-based trace diagnosis

The execution of each rule contributes additional insights into the diagnosis result. For example, a very basic rule identifies the slowest method (i.e., the one with the highest response or exclusive time) in a trace. Next we need to obtain more details on the problem, e.g., "Is it just one method invocation or an invocation of multiple methods?", "How often is it invoked/are they invoked?", "Are remote calls involved?". In our example, this includes the analysis of response times and exclusive response times of the methods invoked from *searchTitle(String)*, i.e., a set of *executeQuery()* methods. Among others, the rule for the "N+1" problem will be executed subsequently. It will obtain and analyze the statements executed by these methods. If they show the typical N+1 symptoms, *diagnoseIT* will report this and suggest how it can be solved. In case respective information is available, domain- and technology-specific rules are executed to semantify the root cause. For example, there can be a one-to-many object relationship in the monitored application between this one object and a collection of objects, that is lazy-loaded by an O/R mapper such as Hibernate. In case additional information about technologies (e.g., O/R libraries) is made available by the underlying APM tool, *diagnoseIT* can provide recommendations on how to optimize their respective configurations. Note that no state information is kept after the analysis of a trace, but *diagnoseIT* can group traces that have similar results to reduce analysis report size.

Rules, presented here using Drools[2] engine syntax, follow the schematic pattern shown in the listing (exemplified for the N+1 problem):

```
rule "N+1 problem"
when
    traceTaggedWithManySQLCalls()
then
    if(analyzeForNPlusOne()) addTagNPlusOne();
```

Rules are executed on the provided trace as long as the trace satisfies the *when* clause in one of the rules. Every time some rule finds a symptom in the trace, it will apply a tag to it. This ensures that the analysis will progress without executing the same rules again, avoiding detecting already detected symptoms. In this sample, if the trace contains many database calls, they will be analyzed. If they conform to the common N+1 anti-pattern structure, the tag for N+1 will be applied.

IV. RELATED WORK

Related work comes from the field of automatic performance problem diagnosis, including both model-based and measurement-based approaches, as well as diagnosis features implemented in commercial APM tools.

A couple of works try to detect performance anti-patterns [17], [18]. Trubiani et al. [19] formalize performance anti-patterns as a set of rules, referring to elements in architectural performance modeling languages and try to detect these anti-patterns by analyzing models and prediction results. Similarly, Parsons and Murphy [15] try to detect anti-patterns based on models extracted from runtime measurements. Their work is limited to Java EE systems. Peiris and Hill [16] present an approach which uses system performance metrics to detect the one-lane bridge anti-pattern [17], but it does not provide the root-cause. Wert et al. [21] propose to systematically perform experiments to search for performance anti-patterns

[2]Drools, http://www.drools.org/

in software. The approach uses a decision tree to search for anti-patterns based on symptoms detected in executed load tests. The mentioned approaches focus either on architectural performance problems or have not been designed for production scenarios.

A couple of other approaches have been proposed to detect performance problems during the testing phase. Problem detection in measurement-based tools is often performed by comparing the obtained data to manually defined thresholds or thresholds created by the tool as a baseline. Jiang et al. [10] compare results from load tests to a predefined baseline to detect anomalies. Grechanik et al. [8] reduce the number of test cases using machine learning techniques to select only those tests that are meaningful for performance problem detection. Based on results of these selected tests, they detect performance bottlenecks. There are measurement-based approaches that use regression testing to detect anomalies [6], [7], but they do not focus on root cause analysis.

Various commercial (e.g., CA APM, Dynatrace, AppDynamics [11]) and open-source (e.g., [13], [20]) APM tools exist. They usually support the detection of performance problems, e.g., using baselines based on historical data or manually defined thresholds, as well as alerting features with only limited support for automatic diagnosis. Some newer tools going into a direction to *diagnoseIT* have been announced [1], [2]. However, to the best of our knowledge they do not provide any semantification of problems and their analysis strategies are not extensible.

V. Conclusions and Outline

Current APM practice is still determined by various manual and recurring tasks, particularly when it comes to the configuration and maintenance of APM infrastructures as well as the diagnosis of performance problems. In this paper, we have presented the preliminary work on our *diagnoseIT* approach for expert-guided automatic diagnosis of performance problems in enterprise applications. We are currently pursuing this research in a collaborative project [14], with academic and industrial partners. The current focus is on the trace-based automatic diagnosis of performance problems based on existing works on anti-pattern detection. In the near future, we will focus on the inclusion of diagnosis strategies based on adaptive instrumentation and, along with that, on the development of efficient techniques that manage a trade-off between measurement detail and system perturbation (IQM). In a later stage, we plan to include system performance measures, e.g., CPU utilization. In addition to the outlined functional goals, we plan to foster the interoperability between APM vendors. One such initiative is an execution trace interchange format. In addition to lab experiments with distributed EAs, we will evaluate our approach on large-scale EAs provided by the two associated industrial consortium partners. The developed methods, techniques, and tool will be published under an open-source license [14]. Currently, the working prototype, integrated into the inspectIT APM tool, is available.

VI. Acknowledgements

This work is being supported by the German Federal Ministry of Education and Research (grant no. 01IS15004, diagnoseIT) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC).

References

[1] Instana. http://www.instana.com/.
[2] Ruxit. http://ruxit.com/.
[3] A. Brunnert et al. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Aug. 2015.
[4] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring stable performance for systems that degrade. In *Proc. 5th Int. Workshop on Software and Performance (WOSP '05)*, pages 43–51, 2005.
[5] A. B. Bondi. *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. Addison-Wesley Professional, 2014.
[6] L. Bulej, T. Kalibera, and P. Tůma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358, May 2005.
[7] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proc. 10th Int. Conference on Quality Software (QSIC 2010)*, pages 32–41, 2010.
[8] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proc. 34th Int. Conference on Software Engineering (ICSE '12)*, pages 156–166. IEEE Press, 2012.
[9] L. Grinshpan. *Solving Enterprise Applications Performance Puzzles: Queuing Models to the Rescue*. Wiley-IEEE Press, 2012.
[10] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *IEEE Int. Conference on Software Maintenance, (ICSM 2009)*, pages 125–134, 2009.
[11] J. Kowall and W. Cappelli. Magic quadrant for application performance monitoring, 2014.
[12] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2002.
[13] NovaTec Consulting GmbH. inspectIT. http://www.inspectit.eu/.
[14] NovaTec Consulting GmbH and University of Stuttgart. diagnoseIT. http://diagnoseit.github.io/.
[15] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–91, 2008.
[16] M. Peiris and J. H. Hill. Towards detecting software performance anti-patterns using classification techniques. *SIGSOFT Softw. Eng. Notes*, 39(1):1–4, Feb. 2014.
[17] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proc. 2nd Int. Workshop on Software and Performance (WOSP '00)*, pages 127–136, 2000.
[18] C. U. Smith and L. G. Williams. More new software antipatterns: Even more ways to shoot yourself in the foot. In *29th International Computer Measurement Group Conference*, pages 717–725, 2003.
[19] C. Trubiani and A. Koziolek. Detection and solution of software performance antipatterns in Palladio architectural models. In *Proc. 2nd ACM/SPEC Int. Conf. on Performance Engineering (ICPE '11)*, pages 19–30, 2011.
[20] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. 3rd ACM/SPEC Int. Conf. on Performance Engineering (ICPE '12)*, pages 247–248, 2012.
[21] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proc. of the 2013 Int. Conf. on Software Engineering (ICSE '13)*, pages 552–561, 2013.