Systematic Guidance in Solving Performance and Scalability Problems

Christoph Heger
Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
christoph.heger@kit.edu

ABSTRACT

The performance of enterprise software systems affects business critical metrics like conversion rate (proportion of visitors who become customers) and total cost of ownership for the software system. Keeping such systems responsive and scalable with a growing user base is challenging for software engineers. Solving performance problems is an error-prone and time consuming task that requires deep expert knowledge about the system and performance evaluation. Existing approaches to support the resolution of performance problems mainly focus on the architecture level neglecting influences of the implementation. In this proposal paper, we introduce a novel approach in order to support software engineers in solving performance and scalability problems that leverages known solutions to common problems. The known solutions are evaluated in the context of the particular software system. As a result, a detailed plan is derived that helps and guides software engineers in resolving the problem. We plan to conduct an industrial case study at SAP.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Measurement, Performance

Keywords

Performance Problem Resolution; Software Engineer Support; Systematic Guidance; Feedback Provisioning

1. INTRODUCTION

The performance (e.g. throughput, response time, resource consumption) of enterprise software systems affects business critical metrics like conversion rates and operational costs, and as such it is critical to the business' success. Software engineers have to ensure high performance and scalability to keep such systems powerful and competitive. In [15], the founders of PayPal and Hotmail report about their large efforts they had to undertake in order to keep their first application versions responsive and scalable with a growing user base. An enterprise software system must be able to continue to meet performance requirements as the load (such as number of users) increases and resources are added. Performance problems are often the root cause of scalability problems. Our scope of performance problems are those that hinder scalability.

Despite all the achievements in the software and performance engineering community, there is only limited support for software engineers in solving performance and scalability problems. Thus, solving such problems is still an error-prone and manual task where software engineers have to answer three major questions: (1) *Which solutions exist for the concrete performance problem?*, (2) *How do the solutions affect the performance of the particular system?*, and (3) *How to implement a given solution?* As a result, software engineers that have little or no experience in performance engineering have to invest a significant amount of time and effort to solve such problems.

There are related approaches in the field of software architecture design, web-frontend optimization and self-* systems (e.g. self-healing, self-adaptive, self-aware). Existing approaches on the architecture-level build on structured problem and solution knowledge and provide automated solution evaluation. However, they neglect important implementation details that often cause performance and scalability problems. Tools like Google PageSpeed [1] are also based on well-known and understood problem and solution knowledge and can automatically apply solutions to identified and known problems. However, they only support software engineers in web-frontend performance optimization. Runtime-based, fully automated approaches for self-* systems often use a control-loop to reactively or proactively adjust or prepare software systems for environmental changes (e.g. workload), optimized resource utilization or fault recovery. However, their goal is not to support software engineers in solving a problem during development.

In this research proposal, we introduce a novel approach to support software engineers in solving performance and scalability problems based on known solutions for common problems. Our goal is to provide systematic guidance for software engineers in problem resolution. We build on struc-

WCOP'13, June 17, 2013, Vancouver, BC, Canada.

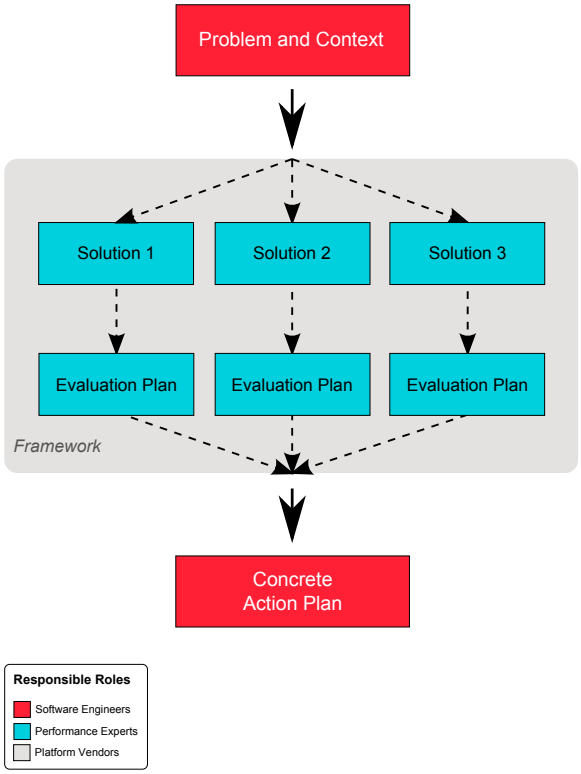

Figure 1: Idea of the approach.

tured problem and solution knowledge to evaluate known solutions for the particular software system based on systematic measurements. We begin with looking into software bottlenecks (e.g. resource pools) that can hinder the scalability of the targeted software systems. We strive for generic applicability of our proposed approach but our research efforts will mainly focus on Java-based software systems, because we plan to conduct an industrial case study at SAP in the context of Java-based enterprise cloud applications.

The research proposal is structured as follows: In Section 2, we describe the idea of our approach that drives our research. In Section 3, we introduce the challenges we have to overcome that come with our idea. In Section 4, we present our envisioned approach for systematic guidance of software engineers, including our generic process, evaluation plans for solution evaluations and action plans to support software engineers in applying the solution. In Section 5, we present and discuss related work. In Section 6, we describe our planned validation before closing the proposal with our contribution in Section 7.

2. IDEA

The goal of our research is to support software engineers in solving scalability problems based on established solutions for known problems. For this purpose, we need to identify the solution space for concrete problems, suitable solutions in the context of a particular software system, and actions for software engineers to apply the solution. We assume as context, that a performance problem is discovered in a software system (that is an obstacle for scalability) and software engineers are assigned with the task to solve the problem.

Our idea involves three different roles, namely software engineers, performance experts and platform vendors (cf. Figure 1). Software engineers are the target audience. They contribute the *Performance Problem* and its *Context* (e.g. root cause). Furthermore, they are responsible for trade-off decisions and apply the *Solution* to resolve the problem. Performance experts contribute structured problem and solution knowledge as well as *Evaluation Plans* describing requirements and processes to evaluate the applicability of solutions. Platform vendors provide a framework to software engineers for systematic guidance in solving performance problems. The framework supports software engineers in identifying and evaluating possible solutions. It can be shared across platform vendors providing (platform specific) solutions and evaluation plans. The benefit of our novel idea is the guidance of software engineers in solving performance problems, that are inexperienced in software performance engineering, with a systematic process based on expert knowledge of performance experts. Based on a given concrete performance problem, there are often different solutions possible, but not all will be applicable and solve the performance problem in a specific case. Therefore, our approach searches the solution space, executes evaluation plans, makes suggestions for suitable solutions and provides action plans for software engineers. The main contribution of the framework is guidance through the problem solving process and support in (semi-) automated evaluation of solutions and derivation of *Concrete Action Plans*.

We illustrate our idea with an example. We suppose that software engineers have discovered a software performance bottleneck manifested in the resource pool for database connections in a benchmark application. The known solutions of performance experts are (1) to increase the amount of resources available in the connection pool, (2) to replace the connection pool implementation, or (3) to reduce holding times of database connections. The framework uses the evaluation plan of each solution to evaluate the applicability and resolution of the problem in this specific case. It executes systematic experiments and varies the connection pool parameters in (1), predicts the performance of alternative implementations in (2), and identifies code statements that can be moved to reduce holding times in (3). The results are presented to software engineers that know the software system best and are then able to make trade-off decisions if necessary. The tooling then derives the actions for problem resolution for the selected solution(s) to guide software engineers.

3. CHALLENGES

In this Section, we describe the challenges that come with our idea and are addressed with our research:

Development of a generic process for tool supported, systematic performance problem solving. Software engineers that have only little or no experience in software performance engineering have to invest a significant amount of time and effort to solve performance problems. We have to provide a generic systematic process to support them in solving such problems. The goal of the process is to support software engineers by providing a guided, systematic identification of a solution. To lower the burden for software engineers, we have to design the process to be easily understandable and applicable without requiring extensive

training and expert knowledge. We have to identify required tools that are available or must be developed and we have to investigate which parts of the overall process can be automated.

Creating a catalogue of problems and solutions. Problem and solution knowledge is spread over many different sources, unorganized and unstructured, making it hard to identify the solution space of a concrete performance problem. The knowledge about how to solve common performance problems are often the result of many years of experience of experts and best practices. The software performance engineering community has started to identify performance antipattern [3, 10, 11, 23, 24] describing common recurring problems and their solutions. Software engineering experts, working for many years in industry, published books [12, 22] about performance tuning of Java-based software systems. We have to organize and structure available knowledge by creating a *Problem and Solution Catalogue* to identify the solution space for a known performance problem with less effort.

Development of an Evaluation Plan for each solution. Based on the concrete performance problem, there are often different solutions possible, but not all will be applicable or solve the problem in every case. The evaluation of different solutions can require different approaches and tools. For example, one solution can require a measurement-based evaluation, while others need statistical models for prediction or architectural models. We need an *Evaluation Plan* from performance experts describing what has to be done in order to evaluate a solution. The goal of each *Evaluation Plan* is to be realizable, (semi-) automatable and being less effort for software engineers. For this purpose, we have to provide a systematic process to enable performance experts the development of *Evaluation Plans* for solutions. We have to draw the conclusion if a solution is applicable and solves the problem from evaluation results. For this reason, performance experts need a systematic process for the development of heuristics, to validate and understand their limitations and to compare the performance of different heuristics to select the strongest.

Execution of an Evaluation Plan. The execution of systematic experiments, performance prediction or static code analysis have all their complexity. We need to provide tool supported (semi-) automated execution of *Evaluation Plans* to be less effort for software engineers. For this purpose, we have to identify the requirements and develop tooling that enables (semi-) automated execution. For example, a test system and a test scenario (e.g. usage profile, workload) have to be available to run experiments.

Derivation of an Action Plan for software engineers. Based on the concrete *Evaluation Plan*, there are often changes to the software system necessary. Furthermore, after a suitable solution is identified software engineers have to apply the solution. For this purpose, we need *Action Plans* from performance experts guiding software engineers in changing the software system. Our goals for an *Action Plan* are to be easy understandable, applicable and tailored to the task and context. We have to identify which additional information is required to derive a *Concrete Action Plan*. Moreover, we have to understand to which degree an *Action Plan* is automatically applicable. While there are many degrees of freedom involved, we have to clearify in which situations a *Concrete Action Plan* cannot be derived.

4. APPROACH

In this Section, we present our envisioned process for systematic performance problem solving. Figure 2 gives an overview of the process. The process consists of three process steps: *Solution Space Identification*, *Solution Evaluation*, and *Action Plan Customization*.

The *Problem and Solution Catalogue (PSC)* and the concrete *Performance Problem* are initial input to the process. PSC is a structured representation of the knowledge about common problems and their solutions. It can be compared with guides for doctors describing causes, symptoms, history and treatment of deseases. The doctor in our case is the software engineer and the patient is the software system. The given *Performance Problem* can describe root causes and symptoms. In the *Solution Space Identification* step, the goal is the identification of possible solutions for a concrete performance problem in a given context. For this purpose, we look up PSC for problems that match the given problem of the system. We compare the known symptoms and root causes with the characteristics of known problems. This is similar to a doctor researching literature to identify deseases matching the observed symptoms. The result of this step is a set of solutions that are known as treatments for the given problem. Like in medicine, there are potentially different treatments, however, not all will be always applicable and heal the desease in any given case. The doctor uses the medical history of his patient and further medical examinations to identify the most suitable treatment.

In the *Solution Evaluation* step, the goal is the identification of suitable solutions for a concrete software system. We use *Evaluation Plans* to evaluate different solutions and to identify the most suitable one. This is comparable with further medical investigations. Like in medicine, there are different requirements for a medical examination such as medical examination methods, medical devices as well as the presence of the patient in most cases. For this purpose, *Evaluation Plans* describe the process how a particular solution has to be evaluated. Furthermore, they specify the functional requirements (e.g. test system availability). *Generic Action Plans* describe what needs to be changed in the software system. Depending on the concrete solution, the evaluation can be based on different approaches and tools. For example, a measurement-based approach can involve the execution of systematic experiments. Therefore, we use the provided *Test System* and *Test Scenario* to run experiments. The evaluation can also be model-based, requiring architectural models, and can involve simulation. We use heuristics to evaluate the applicability of each solution. The framework presents applicable solutions to the software engineers and ranks them by expected performance improvement. Software engineers can then make trade-off decisions, if necessary, and select the solution(s) they want to apply.

In the *Action Plan Customization* step, the goal is to derive steps for software engineers to apply a solution. This can be compared with the doctor selecting the medication and the dose for his patient based on the treatment. The steps of a *Concrete Action Plan* describe how a solution is applied to a particular software system. For this purpose, the *Concrete Action Plan* is then derived for a specific case

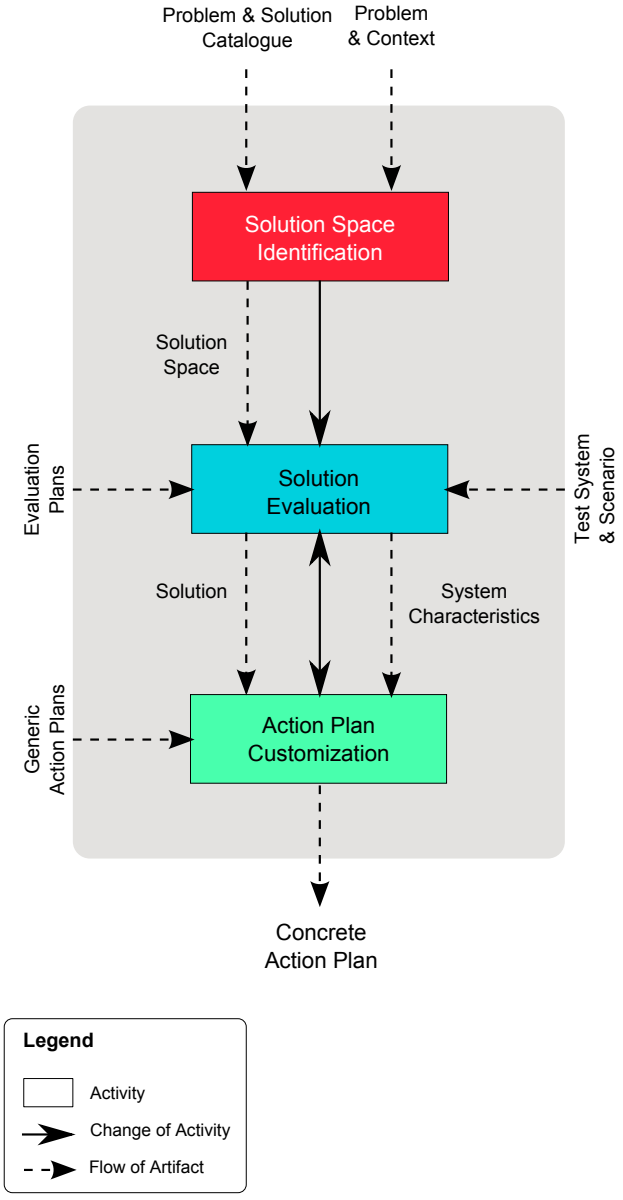

Figure 2: Process overview.

by customizing the *Generic Action Plan* with information about the software system and presented to the software engineers. For example, statements and configuration parameter names can be derived from the implementation and included in the plan.

5. RELATED WORK

In this section, we discuss related work of the fields feedback provisioning, web-frontend optimization and self-* systems. We also present complementary work to our idea.

The research area of guiding engineers in the selection of an appropriate solution when different alternatives for the same problem exists is known with the term *feedback provisioning*. There are already approaches in literature coping with feedback. Examples are the ruled-based approaches described in [20, 27, 7, 9], the meta-heuristic approaches described in [17, 4, 2], or the design space exploration approaches described in [21, 13, 18]. Drago et al. present in [9] an extension of the QVT-Relations language with contructs to express design alternatives, their impact on non-functional metrics like performance, and how to evaluate them and guide a non-expert engineer in selecting the most appropriate solution. In [27], Jing Xu presents an approach for rule-based diagnosis of performance problems that triggers recommendations for improvements. The improvements address runtime configuration of the software system or software design itself. The recommendations are derived from published methodology of software performance engineering. The author presents a tool called Performance Booster that is derived from a systematic improvement process. Cortellessa and Trubiani present a model-based approach [8] to estimate the performance of security solutions. They introduce rules that drive the composition of security services performance models. Their goal is to support designers in decisions related to the security vs. performance trade-off. However, all approaches are model-based and neglect important implementation details.

Nguyen et al. present an approach [19] to support software engineers in applying bug fixes. Their goal is to recommend relevant, useful code changes based on the knowledge about the software system and its recurring bug fixes. For this purpose, their tool called FixWizard uses heuristics to identify similar code fragments and their fixes in the past. However, their approach only addresses functional faults neglecting performance.

To improve the deployment of components, Malek et al. introduce a framework [16] that guides the developer in the design of their solutions for component redeployment for large distributed systems. The goal is to find a deployment architecture that exhibits desirable system characteristics or satisfies a given set of constraints. They use runtime monitoring and consider quality of service (e.g. latency, availability). However, the goal is not to solve performance problems manifested in the software system.

In the domain of self-* systems, Carzaniga et al. present an approach [6] for the automated application of workarounds in web applications for functional faults at runtime. They use knowledge about known problems and workarounds to derive program-rewriting rules. In [5], they outline the same idea for Java applications. However, their focus is on functional faults and runtime.

Complementary to our idea is the approach [25] to discover performance problems with systematic experiments

based on a framework called Software Performance Cockpit[1] [26]. The authors established a desicion tree that enables them to detect performance problems and identify the root cause by observing certain patterns in the behaviour of the application (e.g. throughput, response time, resource utilization). For the architecture-level, PerOpteryx [14] represents a promising approach for automated performance optimization of architectural models. The authors apply performance tactics and meta-heuristics to find pareto-optimal solutions for performance problems in architecturel models.

6. PLANNED VALIDATION

We plan to validate our approach with at least three case studies. The goals of each are to answer three questions: (1) *Are the solutions appropriate for the software system?*, (2) *Is the provided guidance to apply the solution to the software system applicable for the software engineer?*, and (3) *How good is the achieved performance improvement?* In the case of (3), we plan to measure the difference (e.g. throughput, response time, resource utilization) before and after the resultion.

We further plan to use case study results to discuss the effort with and without our approach for software engineers in terms of code changes, conducted experiments as well as for performance experts to solve a performance problem. We aim at understanding if the effort for performance experts for single definition of solutions and associated evaluation plans are out balanced by the possible reuse by many software engineers.

We plan the validation in three phases: In the first case study, we plan to use fault injection techniques on a well known software system (e.g. benchmark application) to hinder scalability in order to evaluate our approach. Moreover, in the second case study, we plan to use already solved scalability problems with the goal to identify at least the same or an even better solution but with less effort. Finally, we plan to apply our approach in an industrial case study at SAP in the context of enterprise cloud applications.

7. CONTRIBUTION AND PROGRESS

Overall, we identify the following contributions for the proposed PhD thesis:

1. A generic process for tool supported, systematic performance problem solving.
2. A problem and solution catalogue structuring expert knowledge on software bottlenecks and their solutions.
3. A framework for platform vendors providing our process to software engineers.
4. Instantiation of the framework at SAP for software bottlenecks including the development of evaluation and action plans.

As next steps, we plan to create a proof of concept and apply our approach to a known performance problem. We further start to gather information about common performance problems and their solutions with respect to software bottlenecks with the goal of identifying the most common performance problems and to develop evaluation plans.

[1]Available at http://www.sopeco.org

8. ACKNOWLEDGMENTS

I would like to thank Jens Happe, Dennis Westermann, Roozbeh Farahbod, and Alexander Wert for their constructive comments and feedback. I would also like to thank Prof. Dr. Ralf H. Reussner for supervising this thesis. This work is supported by the German Research Foundation (DFG), grant RE 1674/6-1.

9. REFERENCES

[1] Google PageSpeed, 2013. `https://developers.google.com/speed/pagespeed/`.

[2] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *Proceedings of the 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '09, pages 61–71, Washington, DC, USA, 2009. IEEE Computer Society.

[3] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[4] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[5] A. Carzaniga, A. Gorla, A. Mattavelli, and N. Perino. A self-healing technique for java applications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1445–1446, Piscataway, NJ, USA, 2012. IEEE Press.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 237–246, New York, NY, USA, 2010. ACM.

[7] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani. A process to effectively identify "guilty" performance antipatterns. In *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 368–382, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] V. Cortellessa and C. Trubiani. Towards a library of composable models to estimate the performance of security solutions. In *Proceedings of the 7th international workshop on Software and performance*, WOSP '08, pages 145–156, New York, NY, USA, 2008. ACM.

[9] M. Drago, C. Ghezzi, and R. Mirandola. A quality driven extension to the qvt-relations transformation language. *Computer Science - Research and Development*, pages 1–20, 2011.

[10] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf. *J2EE antipatterns*. Wiley, 2003.

[11] R. Dugan Jr, E. Glinert, and A. Shokoufandeh. The sisyphus database retrieval software performance antipattern. In *WOSP*, pages 10–16. ACM, 2002.

[12] C. Hunt and B. John. *Java Performance*. Addison-Wesley, 2011.

[13] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 39–48, New York, NY, USA, 2010. ACM.

[14] A. Koziolek, H. Koziolek, and R. Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 33–42, New York, NY, USA, 2011. ACM.

[15] J. Livingston. *Founders at work: stories of startups' early days*. Apress, Berkeley, CA, 2008.

[16] S. Malek, M. Mikic-Rakic, and N. Medvidovic. An extensible framework for autonomic analysis and improvement of distributed deployment architectures. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, pages 95–99, New York, NY, USA, 2004. ACM.

[17] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 105–116, New York, NY, USA, 2010. ACM.

[18] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[19] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA, 2010. ACM.

[20] T. Parsons. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In *Proceedings of the 2nd international doctoral symposium on Middleware*, DSM '05, pages 1–5, New York, NY, USA, 2005. ACM.

[21] T. Saxena and G. Karsai. Mde-based approach for generalizing design space exploration. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 46–60, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] J. Shirazi. *Java Performance Tuning: Efficient and Effective Tuning Strategies*. O'Reilley, 2000.

[23] B. Smaalders. Performance anti-patterns. *Queue*, 4(1):44–50, 2006.

[24] C. Smith and L. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *CMG-CONFERENCE-*, volume 2, pages 667–674, 2003.

[25] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*, ICSE '13, New York, NY, USA, 2013. ACM.

[26] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '10, pages 31–38, Washington, DC, USA, 2010. IEEE Computer Society.

[27] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *Proceedings of the 7th international workshop on Software and performance*, WOSP '08, pages 1–12, New York, NY, USA, 2008. ACM.