Towards the Identification of "Guilty" Performance Antipatterns

Vittorio Cortellessa[⋆], Anne Martens[†], Ralf Reussner[†], Catia Trubiani[⋆]
[⋆]Università degli Studi dell'Aquila, L'Aquila, Italy
Email: {vittorio.cortellessa,catia.trubiani}@univaq.it
[†]Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
Email: {martens,reussner}@ipd.uka.de

ABSTRACT

The problem of interpreting the results of software performance analysis is very critical. Software developers expect feedback in terms of architectural design alternatives (e.g., re-deploy a component), whereas the results of performance analysis are pure numbers. Support to the interpretation of such results that helps to fill the gap between numbers and software alternatives is still lacking. Performance antipatterns can play a key role in the search of performance problems and in the formulation of their solutions. In this poster, we introduce a process to elaborate the analysis results and to score performance requirements, model entities and "guilty" performance antipatterns.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics – *performance measures*; C.4 [Computer Systems Organization]: Performance of Systems – *modeling techniques* **General Terms:** Design, Performance

1. INTRODUCTION

The problem of interpreting results of performance analysis and providing feedback to software designers to overcome performance issues is probably the most critical open issue in the field of software performance engineering today. A large gap in fact exists between the representation of analysis results and the feedback expected by software designers. The former usually contains numbers (such as mean response time and throughput variance), whereas the latter should embed architectural design suggestions useful to overcome performance problems (such as modifying the deployment of certain software components).

A consistent effort has been made in the last decade to introduce automation in the generation of performance models from software models [1], whereas the reverse path from analysis results back to software models is still based on the capabilities of performance experts to observe the results and produce solutions. Automation in this path would help to introduce performance analysis as an integrated activity in the software life cycle, without dramatically affecting the daily practices of software engineers. Strategies to drive the identification of performance problems and to generate feedback on a software model can be based on different elements that may depend on the adopted model notation, on the application domain, on environmental constraints, etc.

2. ANTIPATTERNS-BASED PROCESS

Our approach rests on the capability to automatically detect and solve *performance antipatterns*. In general antipatterns document common mistakes ("bad practices") made during software development as well as their solutions: what to avoid and how to solve the problems. In particular, performance antipatterns [7] describe recurring software performance problems and their solution. Examples presented in [7] are "Circuitous Treasure Hunt" and "The One Lane Bridge".

Figure 1 shows the process we propose: the goal is to modify a software system model in order to produce a new model where the performance problems of the former one have been removed. Boxes in the figure represent data, and segments represent steps.

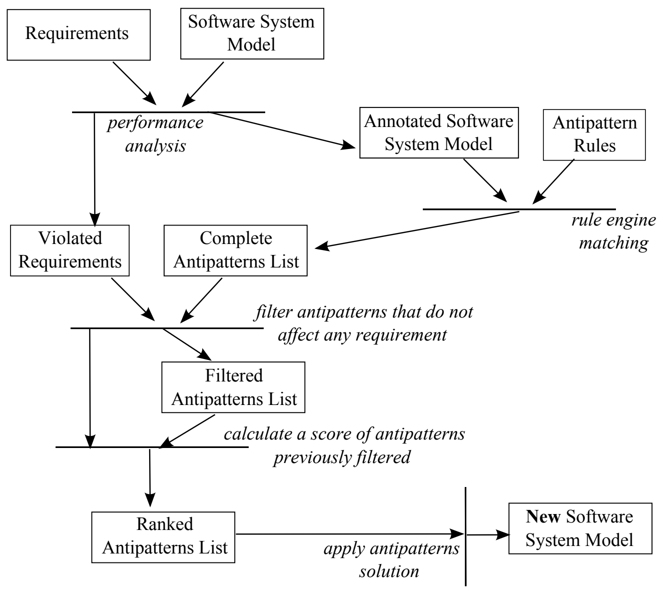
Requirements
Software System Model
performance analysis
Annotated Software System Model
Antipattern Rules
rule engine matching
Violated Requirements
Complete Antipatterns List
filter antipatterns that do not affect any requirement
Filtered Antipatterns List
calculate a score of antipatterns previously filtered
Ranked Antipatterns List
apply antipatterns solution
New Software System Model

Figure 1: Performance analysis interpretation.

The inputs of our process are: a *software system model* and a set of performance *requirements*. The software system model is a model of software and hardware architecture of the current system at hand. It contains all information required for an automated transformation into a performance analysis model, that basically is: resource demands of software tasks, control flow, allocation of software tasks to hardware processors, workload, and operational profile of the system. The requirements represent what end-users and administrators expect from the system and thus represent the target performance properties to be fulfilled.

First, the performance indices of the current software system model are determined in a *performance analysis* step. For example, response time and throughput values for services offered by the system are determined. We obtain two types of results from this step. First, we obtain an *annotated system model*, which is the current software model annotated with performance results. Second, we can check the requirements that are fulfilled by the current software system and create a list of *violated requirements*. If no requirements is violated by the current software system then the process terminates here.

Antipattern rules represent an input that enters the process at the second step. They formalise known performance antipatterns so that they can be automatically detected by a rule engine (see, for example, [8, 6]). Antipattern rules are applied to the annotated model to detect all performance antipatterns and list them in a *complete antipatterns list*.

Then we compare the complete antipatterns list with the violated requirements and examine relationships between detected antipatterns and each violated requirement through the system entities involved in them. We obtain a *filtered antipatterns list*, where antipatterns that do not affect any violated requirement have been filtered out.

In the next step, on the basis of relationships observed before, we estimate how guilty an antipattern is with respect to a violated requirement by calculating a heuristic guiltiness score. As a result, we obtain a *ranked antipatterns list* for each violated requirement.

Finally, a new improved software system model can be built by applying to the current software system the solutions of one or more high-ranked antipatterns for each violated requirement. This last process step can be quite complex, as it can require several iterations to identify the best combination of antipatterns to solve.

3. RELATED WORK

In this section we discuss the related work that deals with automated approaches to improve the performance of software systems based on analysis results.

Xu et al. [8] present a semi-automated approach to find configuration and design improvement on the performance model level. Two types of performance problems are identified in a first step: bottleneck resources and long paths. Then, rules containing performance knowledge are applied to solve the detected problems. The approach uses a depth-first search to try recovery actions from all found performance problems. The approach is notation-specific, because it is based on LQN rules.

Parsons et al. [6] present a framework for detecting performance anti-patterns in Java EE architectures. The method requires an implementation of a component-based system, which can be monitored for performance properties. It uses the monitoring data to construct a performance model of the system and then searches for EJB-specific performance antipatterns in this model. This approach cannot be used for performance problems in early development stages, but it is limited to implemented and running EJB systems only.

Diaz Pace et al. [4] have developed the ArchE framework. ArchE assists the software architect during the design to create architectures that meet quality requirements. It helps to create architectural models, collects requirements and the information needed to analyse the quality criteria for the requirements, provides the evaluation tools for modifiability or performance analysis, and suggests improvements. Currently, only rules to improve modifiability are supported. A simple performance model is used to predict performance metrics for the new system with improved modifiability.

In our previous work [3], we have proposed an approach for automated feedback generation for software performance analysis that aims at systematically evaluating performance prediction results using step-wise refinement. The approach relies on the manual detection of performance antipatterns in the performance model. There is no support to rank and solve antipatterns. More recently, in [2] we have presented an approach to automatically detect performance antipatterns based on a formal description and model-driven techniques, which could be used for the "rule engine matching" step in Figure 1.

In another previous work we have proposed a complementary approach to improve software performance for component-based software systems based on metaheuristic search techniques [5]. We proposed to combine random moves and heuristic rules (such as presented here) to search the given design space.

4. CONCLUSION

This poster paper presents the idea of a process addressing the problem of ranking possible design alternatives (represented by antipattern solutions) in order to identify the ones that better address the system flaws emerged from the performance analysis. The process shall help closing the gap in the reverse path from performance analysis results back to choices on the software model level.

5. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[2] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani. Approaching the model-driven generation of feedback to remove software performance flaws. In *Proc. of the 35th Euromicro Conference SEAA-MDD*, 2009. to appear.

[3] V. Cortellessa and L. Frittella. A framework for automated generation of architectural feedback from software performance analysis. In K. Wolter, editor, *Proc. of the 4th European Performance Engineering Workshop (EPEW'07)*, volume 4748 of *LNCS*, pages 171–185. Springer, 2007.

[4] A. Díaz Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In S. Becker, F. Plasil, and R. Reussner, editors, *Proc. of 4th International Conference on the Quality of Software-Architectures (QoSA'08)*, volume 5281 of *LNCS*, pages 171–188. Springer, 2008.

[5] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically improve software models for performance, reliability and cost using genetic algorithms. In *Proc. of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*, New York, NY, USA, 2010. ACM. to appear.

[6] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 2008.

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

[8] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *Proc. of the 7th International Workshop on Software and Performance (WOSP'08)*, pages 1–12, New York, NY, USA, 2008. ACM.