A Process to Effectively Identify "Guilty" Performance Antipatterns *

Vittorio Cortellessa[1], Anne Martens[2], Ralf Reussner[2], Catia Trubiani[1]

[1] Università degli Studi dell'Aquila, L'Aquila, Italy
[2] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{vittorio.cortellessa, catia.trubiani}@univaq.it
{martens, reussner}@ipd.uka.de

Abstract. The problem of interpreting the results of software performance analysis is very critical. Software developers expect feedbacks in terms of architectural design alternatives (e.g., split a software component in two components and re-deploy one of them), whereas the results of performance analysis are either pure numbers (e.g. mean values) or functions (e.g. probability distributions). Support to the interpretation of such results that helps to fill the gap between numbers/functions 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 paper we tackle the problem of identifying, among a set of detected performance antipatterns, the ones that are the real causes of problems (i.e. the "guilty" ones). To this goal we introduce a process to elaborate the performance analysis results and to score performance requirements, model entities and performance antipatterns. The cross observation of such scores allows to classify the level of guiltiness of each antipattern. An example modeled in Palladio is provided to demonstrate the validity of our approach by comparing the performance improvements obtained after removal of differently scored antipatterns.

Key words: Software Performance Engineering, Antipatterns, Feedback, Performance Analysis

1 Introduction

The problem of interpreting the results of performance analysis and providing feedback to software designers to overcome performance issues is probably the most critical open issue today in the field of software performance engineering. 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).

* This work has been partly supported by the italian project PACO (Performability-Aware Computing: Logics, Models, and Languages) funded by MIUR.

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 provide 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. Our approach rests on the capability to automatically detect and solve *performance antipatterns*. In general, antipatterns [3] document common mistakes (i.e. "bad practices") made during software development as well as their solutions: what to avoid and how to solve the problems. In particular, performance antipatterns [11] describe recurring software performance problems and their solution.

In Figure 1 the process that we propose is reported: 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.

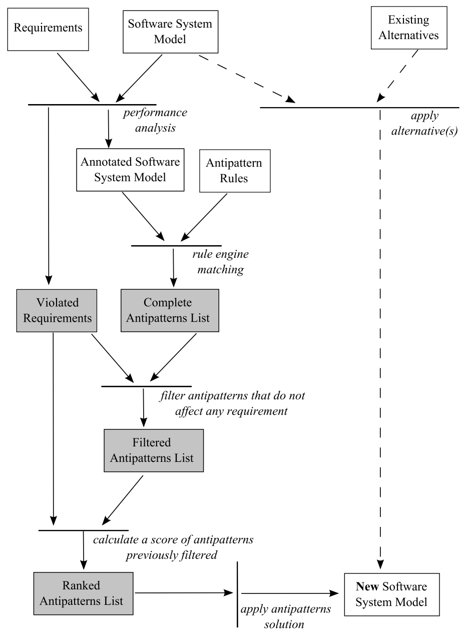
Requirements
Software System Model
Existing Alternatives
performance analysis
apply alternative(s)
Annotated Software System Model
Antipattern Rules
rule engine matching
Violated Requirements
Complete Antipatterns List
filter antipatterns that do not affect any requirement
Filtered Antipatterns List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

Fig. 1. Performance analysis interpretation.

The left hand side of Figure 1, represented with solid arrows, is the part of the process that is based on antipatterns and is the object of this paper. Other techniques can be used to solve performance problems and are represented on the right hand side of Figure 1. In this side a list of *existing alternatives* can contain a number of options for what could be changed in the software system model. From this list, alternatives can be chosen to directly create a new software system models [9]. Alternatives to apply can be chosen randomly, manually by software architects based on their experience, or based on heuristics. These techniques are out of the scope of this paper, thus the right-hand path is represented with dashed arrows in Figure 1.

The inputs of our process are: a *software system model* and a set of performance *requirements*. The software system model contains all information required for an automated transformation into a performance analysis model, that basically is: resource demands of software services, control flow, allocation of software services to hardware processors, workload and operational profile of the system. The requirements represent what end-users 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. We obtain two types of results from this step: (i) an *annotated system model*, which is the current software model annotated with performance results, and (ii) a list of *violated requirements* as resulting from the analysis. If no requirement is violated by the current software system then the process terminates here. *Antipattern rules* represent a system-independent 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, [10, 12])([1]). 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. We obtain a *filtered antipatterns list*, where antipatterns that do not affect any violated requirement have been filtered out. In the following step, on the basis of relationships observed before, we estimate how guilty an antipattern is with respect to a violated requirement by calculating a 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.

In this paper we focus on the process steps that take place between the shaded boxes of Figure 1. We assume that the performance analysis of the initial model has identified a list of violated requirements, and we also assume that a rule engine has parsed the current software system model to build a complete antipatterns list [4]. The questions tackled in this paper are the following: (i) "What are the most guilty antipatterns?" and (ii) "How much does each antipattern contribute to each requirement violation?". The contribution of this paper is a technique to rank antipatterns in the model on the basis of their guiltiness for violated requirements. Such ranked list will be the input to the solution step that can use it to give priorities to certain antipattern solutions. Without such ranking technique the antipattern solution process can only blindly move among antipattern solutions without eventually achieving the desired result of requirements satisfaction.

The paper is organized as follows. Section 2 describes our approach to the antipattern ranking, in section 3 we illustrate the application of our approach to a case study (i.e. a web reporting system) in the Palladio Component Model (PCM) [2], Section 4 focuses on the open issues of the proposed approach, Section 5 presents the related work, and finally in Section 6 conclusions are provided.

[1] We have introduced a technique based on first-order logic to specify such rules [5].

2 Our approach for antipattern ranking

In this section we provide a detailed description of our approach shown in the shaded boxes of Figure 1. The input data for our approach are a set of *violated requirements* (Section 2.1) and a *complete antipatterns list* for the system under study (Section 2.2). In the first step, we filter out antipatterns that do not affect any requirements and obtain a matrix of *filtered antipatterns* (Section 2.3). In the second step, we assign a guiltiness score for the filtered antipatterns with respect to each violated requirement (Section 2.4). The resulting *ranked antipatterns list* for each requirement can be used to decide which antipattern solution(s) to apply in order to obtain an improved software system model.

2.1 Violated Requirements

The performance requirements that, upon the model analysis, result to be violated represent very likely the effects (to be removed) of some antipatterns, therefore we focus on them.

System requirements are classified on the basis of the performance indices they address and the level of abstraction they apply. Here we consider requirements that refer to the following performance indices [8]([2]):

- *Response time* is defined as the time interval between a user request of a service and the response of the system. Usually, upper bounds are defined in “business” requirements by the end users of the system.
- *Utilisation* is defined as the ratio of busy time of a resource and the total elapsed time of the measurement period. Usually, upper bounds are defined in “system” requirements by system engineers on the basis of their experience, scalability issues, or constraints introduced by other concurrent software systems sharing the same hardware platform.
- *Throughput* is defined as the rate at which requests can be handled by a system, and is measured in requests per time. Throughput requirements can be both “business” and “system” requirements, depending on the target it applies; for the same motivation it can represent either an upper or a lower bound.

Various levels of abstraction can be defined for a requirement: system, processor, device (e.g., CPU, Disk), device operation (e.g., read, write), software component, basic and composed services. In the following, by “basic service” we denote a functionality that is provided by a component without calling services of other components. By “composed service”, we denote a functionality that is provided by a component and involves a combination of calls to services of other components. Both types of services can be offered to the end user at the system boundary, or be internal and only used by other components.

However, we do not consider all possible combinations of indices and levels of abstraction. Our experience on system requirements leads us to focus on the most

[2] Note that the values of all these indices depend on the system workload.

frequent types of requirements, that concern: *utilisation* of processors, *response time* and/or *throughput* of basic and composed services.

Table 1 contains simplified examples of performance requirements and their observed values. Each requirement is represented by: (i) an identifier (*ID*), (ii) the type of requirement (*Requirement*) that summarizes the performance index and the target system element, (iii) the required value of the index (*Required Value*), (iv) the maximum system workload for which the requirement must hold (*System Workload*), and (v) the observed value as obtained from the performance analysis (*Observed Value*). In Table 1 three example requirements are reported. The first one refers to the utilisation index (i.e., U): it requires that processor $Proc_1$ is not utilised more than 70% under a workload of 200 reqs/sec, while it shows an observed utilisation of 64%. The second one refers to the response time index (i.e., RT) and the third one refers to the throughput index (i.e., T) of certain software services. Requirements R_2 and R_3 are violated, whereas R_1 is satisfied.

ID	Requi-rement	Required Value	System Workload	Observed Value
R_1	U($Proc_1$)	0.70	200 $\frac{reqs}{sec}$	0.64
R_2	RT(CS_y)	2 sec	50 $\frac{reqs}{sec}$	3.07 sec
R_3	T(BS_z)	1.9 $\frac{reqs}{sec}$	2 $\frac{reqs}{sec}$	1.8 $\frac{reqs}{sec}$
...	...	...	...	...

Table 1. Example of Performance Requirements.

ID	Involved Entities
R_2	$Comp_x.BS_a, Comp_y.BS_b, Proc_2$
R_3	$Comp_w.BS_z, Proc_3$
...	...

Table 2. Details of Violated Requirements.

Violated requirements are further detailed by specifying the system entities involved in them. For utilisation requirements, we only consider as involved the processor for which the requirement is specified. For example, if a utilisation requirement has been specified for processing node $Proc_2$, we consider only $Proc_2$ to be involved. For requirements on services (i.e. response time and throughput requirements), all services that participate in the service provisioning are considered as involved. For example, if a violated requirement is specified for a service S_1, and S_1 itself calls services S_2 and S_3, we consider all three services S_1, S_2 and S_3 to be involved. Furthermore, all processing nodes hosting the components that provide involved services are considered as involved ([3]). Namely, if the component providing service S_1 is deployed on a processing node $Proc_1$, and the component(s) providing S_2 and S_3 are deployed on a processor $Proc_2$, we additionally consider $Proc_1$ and $Proc_2$ to be involved. With this definition we want to capture the system entities that are most likely to cause the observed performance problems.

In Table 2, the involved services of two violated requirements are reported: R_2 involves all basic services participating in the composed service CS_y (i.e., BS_a, BS_b) prefixed by the names of components that provide them (i.e., $Comp_x$, $Comp_y$ respectively), whereas R_3 only involves the target basic service BS_z

[3] The allocation of services to processing nodes is part of the Software System Model (see Section 1).

similarly prefixed. The list of involved entities is completed by the processors hosting these components.

2.2 Complete Antipatterns List

We assume that a rule engine has parsed the annotated system model and has identified all performance antipatterns occurring in it. All detected performance antipatterns and the involved system entities are collected in a *Complete Antipatterns List.* An example of this list is reported in Table 3(a): each performance antipattern has an identifier (*ID*), the type of antipattern (*Detected Antipattern*), and a set of system entities such as processors, software components, composed and basic services, that are involved in the corresponding antipattern (*Involved Entities*). In [11] a list of types of antipatterns is reported.

Note that the detection process takes into account only the annotated software system model and the antipattern rules and thus it is independent of the violated requirements.

Table 3. Example: Antipatterns Lists

(a) Complete Antipatterns List

ID	Detected Antipattern	Involved Entities
PA_1	Blob	$Comp_x$
PA_2	Concurrent Processing Systems	$Proc_1$ $Proc_2$
PA_3	Circuitous Treasure Hunt	$Comp_t.BS_z$
...	...	...

(b) Filtered Antipatterns List.

		Requirements			
		R_1	R_2	...	R_j
Anti-patterns	PA_1		$Comp_x$		
	PA_2	$Proc_1$			
	...				
	PA_x				e_1, .., e_k

2.3 Filtering antipatterns

The idea behind the step that filters the list of detected antipatterns is very simple. For each violated requirement, only those antipatterns with involved entities in the requirement survive, whereas all other antipatterns can be discarded.

A *filtered* list is shown in Table 3(b): rows represent performance antipatterns taken from the complete list (i.e. Table 3(a)), and columns represent violated performance requirements (i.e. Table 2). A non-empty (x, j) cell denotes that the performance antipattern PA_x is a candidate cause for the violation of the requirement R_j. In particular, the (x, j) cell contains the intersection set of system entities $\{e_1, .., e_k\}$ that are involved in the antipattern PA_x and the violated requirement R_j. We will refer to this set as $involvedIn(PA_x, R_j)$ in the following. Antipatterns that do not have any entity in common with any violated requirement do not appear in this list.

This filtering step allows to reason on a restricted set of candidate antipatterns for each requirement. In Section 2.4 we illustrate how to use a filtered antipattern list to introduce a rank for each antipattern that allows to estimate its guiltiness vs. a requirement that has been violated.

2.4 Ranking antipatterns

The goal of ranking antipatterns is to introduce an order in the list of filtered antipatterns for each requirement, where highly ranked antipatterns are the most promising causes for the requirement violation. The key factor of our ranking process is to consider the entities involved in a violated requirement. We first assign a score to each entity, and then we rank an antipattern on the basis of a combination of the scores of its involved entities, as follows.

In Table 4 we have summarized all equations that we introduce to assign scores to system entities involved in a violated requirement. As outlined in Section 2.1, the requirements that we consider in this paper are: utilisation of processors, response time and throughput of composed and basic services.

Table 4. How to rank performance antipatterns

Type	Equation
Utilisation	$score_{i,j} = (observedUtil_i - requiredUtil_j)$
Response time	$score_{i,j} = \dfrac{ownComputation_i}{maxOwnComputation_j} \cdot \dfrac{observedRespTime_j - requiredRespTime_j}{observedRespTime_j}$
Throughput	$score_{i,j} = \begin{cases} \frac{requiredThrp_j - observedThrp_j}{requiredThrp_j} & \text{if } workload_i > observedThrp_i \\ & \text{or } isClosed(systemWorkload) \\ 0 & \text{else} \end{cases}$

Utilisation The violation of an utilisation requirement can only target (in this paper scope) a processor. For each violated requirement R_j, we introduce a utilisation score to the involved processor $Proc_i$ as reported in the first row of Table 4. $score_{i,j}$ represents a value between 0 and 1 that indicates how much the $Proc_i$ observed utilisation ($observedUtil_i$) is higher than the required one ($requiredUtil_j$).

Response time The violation of the response time in composed services involves all services participating to that end-user functionality. For each violated requirement R_j, we introduce a response time score to the involved service S_i as reported in the second row of Table 4. We quantify how far the observed response time of the composed service CS_j ($observedRespTime_j$) is from the required one ($requiredRespTime_j$). Additionally, in order to increase the guiltiness of services that mostly contribute to the response time of the composed service, we introduce the first multiplicative factor of the equation. We denote with $ownComputation_i$ the observed computation time of a service S_i participating in the composed service CS_j. If service S_i is a basic service, $ownComputation_i$ equals the response time $RT(S_i)$ of service S_i. However, composite services can also consist of other composite services. Thus, if service S_i is a composite service that calls services S_1 to S_n with probability $P(S_1)$ to $P(S_n)$, $ownComputation_i$ is the response time of service S_i minus the weighted response time of called services:

$$ownComputation_i = RT(S_i) - \sum_{1 \leq c \leq n} P(S_c)RT(S_c)$$

We divide by the maximum own computation over all services participating in CS_j, which we denote by $maxOwnComputation_j$. In this way, services with higher response time will be more likely retained responsible for the requirement violation.

The violation of the response time in basic services involves just the referred service. The same equation can be used, where in this case the first multiplicative factor is equal to 1 as $ownComputation_i$ corresponds to $maxOwnComputation_j$.

Throughput The violation of the throughput in composed services involves all services participating to the end-user functionality. For each violated requirement R_j, we introduce a throughput score to each involved service S_i as reported in the third row of Table 4. We distinguish between open and closed workloads here. For an open workload ($isOpen(systemWorkload)$), we can identify bottleneck services S_i that cannot cope with their arriving jobs ($workload_i > observedThrp_i$). To these services a positive score is assigned, whereas all other services are estimated as not guilty for this requirement violation and a score of 0 is assigned to them. For closed workloads ($isClosed(systemWorkload)$), we always observe job flow balance at the steady-state and thus for all services $workload_i = observedThrp_i$ holds. Thus, we cannot easily detect the bottleneck service and we assign a positive score to all involved services. For the positive scores, we quantify how much the observed throughput of the overall composed service ($observedThrp_j$) is far from the required one ($requiredThrp_j$).

The violation of the throughput in basic services involves just this one service. We can use the previous equation as it is, because the only involved service is the one under stress.

Combining the scores of entities Finally, we rank the antipatterns filtered for each violated requirement R_j. To each antipattern PA_x that shares involved entities with a requirement R_j is assigned a guiltiness degree $GD_{PA_x}(R_j)$ that measures the guiltiness of PA_x for R_j. We consider system entities involved in both PA_x and R_j, as reported in the filtered antipatterns matrix $involvedIn(PA_x, R_j)$. We define the guiltiness degree as the sum of the scores of all involved entities:

$$GD_{PA_x}(R_j) = \sum_{i \in involvedIn(PA_x, R_j)} score_{i,j}$$

Thus the problematic entities that have a high score contribute to consistently raise the overall score of the antipatterns they appear in.

3 Experimenting the approach

In this section we report the experimentation of our approach on a business reporting system case study. First, we describe the example system and the performance analysis with the Palladio approach [2]. Then, we propose the stepwise application of our approach.

3.1 The Business Reporting System (BRS)

The system under study is the so-called Business Reporting System (BRS), which lets users retrieve reports and statistical data about running business processes from a data base. Figure 2 shows an overview of the software system model, visualized in an UML-like diagram, and some labels indicate the detected antipatterns (^{4}).

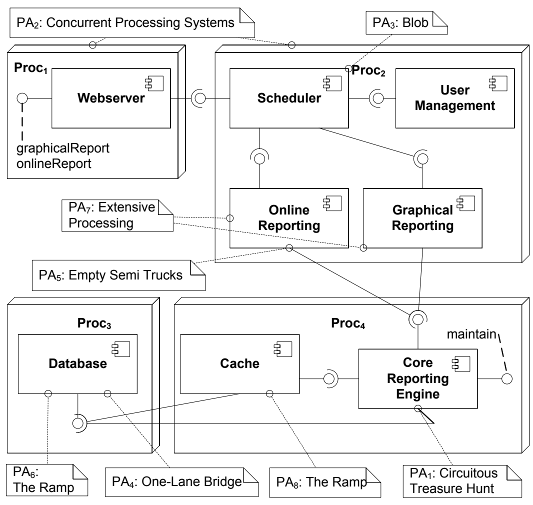

Fig. 2. Case Study: BRS Software System Model

The BRS is a 4-tier system consisting of several software components, as described in the following. The `WebServer` handles user requests for generating reports or viewing the plain data logged by the system. It delegates the requests to a `Scheduler`, which in turn forwards the requests. User management functionality (login/logout) is directed to the `UserMgmt`, whereas report and view requests are forwarded to the `OnlineReporting` or `GraphicalReporting`, depending on the type of request. Both components make use of a `CoreReportingEngine` for the common report generation functionality. The `CoreReportingEngine` accesses the `Database`, for some request types using an intermediate `Cache`.

The system supports seven use cases: users can login, logout and request both reports or views, each of which can be both graphical or online; administrators can invoke the maintenance service. Note that in Figure 2, we only depict those services we specified requirements for.

The PCM model of BRS contains the static structure, the behaviour specification of each component annotated with resource demands and a resource environment specification. For performance analysis, the PCM software system model is transformed automatically into an Extended Queueing Network model suited for simulation with SimuCom [2]. SimuCom is a discrete-event simulator and collects arbitrarily distributed response time, throughput and utilisation for all services of the system.

3.2 Our approach in practice

The results of the performance analysis of the BRS model are reported in Table 5, where the focus is on performance requirements and their observed values. ID’s of violated requirements are typed as bold (i.e. R_2, R_5, R_6, R_7, R_9). In the

ID	Requirement	Required Value	Observed Value
R_1	U($Proc_1$)	0.50	0.08
R_2	U($Proc_2$)	0.75	0.80
R_3	U($Proc_3$)	0.60	0.32
R_4	U($Proc_4$)	0.40	0.09
R_5	RT($CS_{graphicalReport}$)	2.5 sec	4.55 sec
R_6	T($CS_{graphicalReport}$)	0.5 req/sec	0.42 req/sec
R_7	RT($CS_{onlineReport}$)	2 sec	4.03 sec
R_8	T($CS_{onlineReport}$)	2.5 req/sec	2.12 req/sec
R_9	RT($BS_{maintain}$)	0.1 sec	0.14 sec
R_{10}	T($BS_{maintain}$)	0.3 req/sec	0.41 req/sec

Table 5. BRS - Performance requirement analysis

ID	Involved Entities
R_2	$Proc_2$
R_5	$WebServer$, $Scheduler$, $UserMgmt$, $GraphicalReport$, $CoreReportingEngine$, $Database$, $Cache$
R_6	$WebServer$, $Scheduler$, $UserMgmt$, $GraphicalReport$, $CoreReportingEngine$, $Database$, $Cache$
R_7	$WebServer$, $Scheduler$, $UserMgmt$, $OnlineReport$, $CoreReportingEngine$, $Database$, $Cache$
R_9	$CoreReportingEngine$

Table 6. BRS - Violated Requirements

following, we will concentrate on solving the shaded R_5 requirement in order to illustrate our approach.

The violated requirements are further detailed with their involved system entities in Table 6. Following our approach, the detected performance antipatterns occurring in the software system are collected in the *Complete Antipatterns List*, as shown in Table 7. These antipatterns have been also annotated in Figure 2 on the system model.

Table 7. BRS- Complete Antipatterns List

ID	Detected Antipattern	Involved Entities
PA_1	Circuitous Treasure Hunt	Database.getSmallReport, Database.getBigReport $Proc_3$, CoreReportingEngine.getReport, $Proc_4$
PA_2	Concurrent Processing Systems	$Proc_1$, $Proc_2$
PA_3	Blob	Scheduler, $Proc_2$
PA_4	One-Lane Bridge	Database, $Proc_3$
PA_5	Empty Semi Trucks	OnLineReporting.viewOnLine, $Proc_2$, $Proc_4$ CoreReportingEngine.prepareView, CoreReportingEngine.finishView
PA_6	Ramp	Database, $Proc_3$
PA_7	Extensive Processing	GraphicalReporting, OnLineReporting, $Proc_2$
PA_8	Ramp	Cache, $Proc_4$

The combination of violated requirements and detected antipatterns produces the ranked list of BRS antipatterns shown in Table 8. It represents the

[4] All detailed Palladio models and the description of performance antipatterns have been omitted for brevity here, but can be accessed at palladio-approach.net/_AntipatternGuiltiness as well as the configuration of the BRS model (i.e. workload, usage profile, etc.).

result of our antipatterns ranking process, where numerical values are calculated according to the equations reported in Table 4, whereas empty cells contain a value 0 by default, that is no guiltiness.

		Requirements				
		R_2	R_5	R_6	R_7	R_9
Anti-patterns	PA_1		0.558	0.122	0.633	
	PA_2	0.054				
	PA_3	0.054	0.051	0.135	0.032	
	PA_4		0.616	0.161	0.689	
	PA_5	0.054				
	PA_6		0.616	0.161	0.689	
	PA_7	0.054	0.125	0.135	0.06	
	PA_8		0.003	0.015	0.03	

Table 8. BRS - Ranked Antipatterns List

Table 8 can be analyzed by columns or by rows. Firstly, by columns, we concentrate on a certain requirement, for example R_5, and we look at the scores of antipatterns. Our approach indicate which antipatterns are more guilty for that requirement violation (i.e., PA_4 and PA_6) and which is the less guilty one (i.e., PA_8). As another example, four antipatterns affect the requirement R_2, but none of them is apparently more guilty than the other ones. So, in this case our approach is able to identify the antipatterns involved without providing a distinction between them. Yet for the requirement R_9 no detected antipattern has a non-zero guiltiness. This means that the violation of R_9 cannot be associated to any known antipattern. In such a case, further performance improvements could be obtained manually, or the requirement has to be relaxed as it is infeasible.

Observing the table by rows, instead, we can distinguish either the antipatterns that most frequently enter the violation of requirements (i.e. PA_3 and PA_7 in this case) or the ones that sum up to the highest total degree of guiltiness (i.e. PA_4 and PA_6 in this case). Different types of analysis can originate from these different views of the ranked list, however for sake of space in what follows we perform an analysis by columns on requirements R_5 and R_7.

In order to satisfy R_5, on the basis of information in Table 8 we have decided to separately solve one-by-one the following antipatterns: PA_4, PA_6, and as counterexample, PA_8.

PA_4 is a "One-Lane Bridge" in the `Database`. To solve this antipattern, we increase the level of parallelism in the `Database`, thus at the same time multiple threads can access concurrently. PA_6 is a "Ramp" in the `Database`. Here, the data access algorithms have to be optimised for larger amounts of data. This can be solved with a reduced resource demand of the database. In our example, we assumed that the resource demand is halved. PA_8 is a "Ramp" in the `Cache`. The latter accumulates more and more information over time and is slowed down. This can be solved with a reduced resource demand of the `Cache`. In our example, we assumed again that the resource demand is halved.

The results of the ***new*** software systems (i.e., BRS_{PA_x}, the BRS initial system with PA_x solved) are collected in Table 9. It can be noticed that the high guiltness degrees of PA_4 and PA_6 have provided a relevant information because their removal consistently improves the response time. After the removal of PA_8, instead, the requirement R_5 is still violated because it has been removed a cause that affects much less the violated requirement considered.

Table 9. RT($CS_{graphicalReport}$) across different software system models

ID	Requirement	Required Value	Observed Value			
			BRS	BRS_{PA_4}	BRS_{PA_6}	BRS_{PA_8}
R_5	RT($CS_{graphicalReport}$)	2.5 sec	4.55 sec	2.14 sec	2.06 sec	4.73 sec

In Figure 3 we summarize our experiments on the requirement R_5. The target performance index of R_5 (i.e. the response time of the *graphicalReport* service) is plotted on the y-axis, whereas on the x-axis the degree of guiltiness of antipatterns is represented. The horizontal bottommost line is the requirement threshold, that is the response time required, whereas the horizontal topmost line is the observed value for the original BRS system before any modification. Single points represent the response times observed after the separate solution of each performance antipatterns, and they are labeled with the ID of the antipattern that has been solved for that specific point. Of course, the points are situated, along the x-axis, on the corresponding guiltiness degree of the specific antipattern.

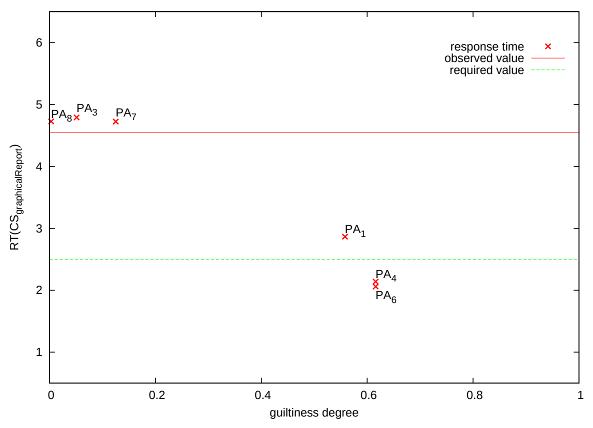

Fig. 3. RT($CS_{graphicalReport}$) vs the guiltness degree of antipatterns.

What is expected to observe in such representation is that the points approaches (and possibly go below) the required response time while increasing their guiltiness degree, that is while moving from left to right on the diagram. This would confirm that solving a more guilty antipattern helps much more than solving a less guilty one, thus validating our guiltiness metric.

All antipatterns with non-zero guiltiness have been solved, one by one, to study their influence on the requirement R_5. Figure 3 very nicely validates our hypothesis, in that very guilty antipatterns more dramatically affect the response time, and their solution leads towards the requirement satisfaction. The same considerations made above can be reiterated for the other requirements ([5]).

[5] Figures summarizing our experiments on requirements R_2, R_6 and R_7 are reported in palladio-approach.net/_AntipatternGuiltiness.

4 Discussion

The experimentation in Section 3 shows promising results for the example that we have considered and the types of requirements introduced. This is a proof of concept that such a ranking approach can help to identify the causes of performance problems in a software system. The experimentation phase has been very important to refine our approach, in fact by observing the performance analysis results we have fine tuned the equations that represent the antipattern ranking.

However, this is only a first step in this direction, and several issues are yet to be addressed. We discuss some of them in this section, as they also represent the main topics of our current and future work in this field.

Refinement of scores and ranking. Although we have obtained promising results in our experiments, the score model can certainly be improved and needs more experimentation on models of different application domains. First, other types of requirements, among the one listed in Section 2, may need appropriate formulas for scoring the entities involved in them. Second, nested requirements could be pre-processed to eliminate from the list of violated requirements those that are dominated from other ones. Third, more experience could lead to refine the antipattern scoring on the basis of, let say, the application domain (e.g. web-based application) or the adopted technology (e.g. Oracle DBMS). For example, a detected “Circuitous Treasure Hunt” might be of particular interest in database-intensive applications, whereas a detected “Concurrent Processing Systems” might be more important for web-based applications. Finally, to achieve more differentiation in the scoring process for guilty performance antipatterns, negative scores to the entities involved in satisfied requirements can be devised.

Lack of model parameters. The application of this approach is not limited (in principle) along the software lifecycle, but it is obvious that an early usage is subject to lack of information because the system knowledge improves while the development process progresses. Lack of information, or even uncertainty, about model parameter values can be tackled by analyzing the model piecewise, starting from complete sub-models. This type of analysis can bring insight on the missing parameters.

Lack of performance indices. In the same situation as above, performance analysis could not produce all indices needed to apply the process. For example, internal indices of subsystems that are not yet designed in details cannot be collected. In this case we can plan a successive (possibly goal-oriented) analysis to collect the lacking performance indices.

Influence of operational profile. Different operational profiles usually give rise to different analysis results that, in turn, may result in different antipatterns identified in the system. This is a critical issue and, as usually in performance analysis experiments, the choice of the operational profile(s) must be carefully conducted.

Influence of other software layers. The performance model that we have considered here only takes into account the software application and the hardware platform. Between these two layers there are other components, such as middleware and operating system, that can embed performance antipatterns.

The approach shall be extended to these layers for a more accurate analysis of the system.

5 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. [12] present a semi-automated approach to find configuration and design improvement on the model level. Based on a Layered Queueing Network model, 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 is notation-specific, because it is based on LQN rules, and also it does not incorporate heuristics to speed-up the search of solutions, as suggested in this paper.

Parsons et al. [10] present a framework for detecting performance antipatterns 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.

Diaz Pace et al. [7] have developed the ArchE framework. ArchE assists the software architect during the design to create architectures that meet quality requirements. 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 [6], we have proposed an approach for automated feedback generation for software performance analysis. 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 [4] we have presented an approach to automatically detect performance antipatterns based on model-driven techniques.

In another previous work we have proposed a complementary approach to improve software performance for component-based software systems based on metaheuristic search techniques [9]. We proposed to combine random moves, as shown on the right hand side of Figure 1, and heuristic rules to search the given design space.

6 Conclusion

In this paper we have shown an approach to rank possible causes of performance problems (i.e. antipatterns) depending on their guiltiness for violated requirements. This work, as shown in Figure 1, is embedded in a wider research area that is the interpretation of performance analysis results and the generation of feedback.

The approach presented here is being integrated with the other work that we have conducted up today in this area. In particular, upstream in Figure 1 we have built a parser (based on XML technologies) that retrieve all antipatterns in a software model. Such parser produces the complete antipattern list that is one of the input of the process presented here. We are tackling the same problem, in parallel, with a model-driven approach. Downstream in Figure 1 we are facing the problem of using the ranked antipattern list to decide the most promising model changes that can rapidly lead to remove performance problems. In this direction several interesting issues have to be faced, such as the simultaneous solution of multiple antipatterns. This research direction can benefit from techniques introduced in model co-evolution.

Finally, we are working to the combination of antipattern-driven approaches, that is the leftmost side of Figure 1, and meta-heuristic approach that run on the rightmost side of the same figure.

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based Performance Prediction in Software Development: A Survey. IEEE TSE. 30(5), 295–310 (2004)
2. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software. 82, 3–22 (2009)
3. Brown, W. J., Malveau, R. C., McCormick III, H. W., Mowbray, T. J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley and Sons (1998)
4. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Approaching the model-driven generation of feedback to remove software performance flaws. In: 35th Euromicro Conference, pp. 162–169. IEEE Press, New York (2009)
5. Cortellessa, V., Di Marco, A., Trubiani, C.: Performance Antipatterns as Logical Predicates. In: 15th International Conference on Engineering of Complex Computer Systems, (to appear). IEEE Press, New York (2010)
6. Cortellessa, V., Frittella, L.: A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis. In: Walter, K. (ed.) EPEW 2007. LNCS, vol. 4748, pp. 171–185. Springer, Heidelberg (2007)
7. Díaz Pace, A., Kim, H., Bass, L., Bianco, P., Bachmann, F.: Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant. QoSA 2008. LNCS, vol. 5281, pp. 171–188. Springer, Heidelberg (2008)
8. Jain, R.: The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley and Sons (1991)
9. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software models for performance, reliability and cost using genetic algorithms. In: Joint WOSP/SIPEW International Conference, (to appear). ACM Press (2010)
10. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Journal of Object Technology. 7(3), 55–90 (2008)
11. Smith, C.U., Williams, L.G.: More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In: Computer Measurement Group Conference (2003)
12. Xu, J.: Rule-based Automatic Software Performance Diagnosis and Improvement. In: Workshop on Software Performance, pp. 1–12. ACM Press (2008)