Performance-Based Software Model Refactoring in Fuzzy Contexts*

Davide Arcelli[1], Vittorio Cortellessa[1], and Catia Trubiani[2]

[1] Università degli Studi dell'Aquila, L'Aquila, Italy
[2] Gran Sasso Science Institute, L'Aquila, Italy
{davide.arcelli,vittorio.cortellessa}@univaq.it
catia.trubiani@gssi.infn.it

Abstract. The detection of causes of performance problems in software systems and the identification of refactoring actions that can remove the problems are complex activities (even in small/medium scale systems). It has been demonstrated that software models can nicely support these activities, especially because they enable the introduction of automation in the detection and refactoring steps. In our recent work we have focused on performance antipattern-based detection and refactoring of software models. However performance antipatterns suffer from the numerous thresholds that occur in their representations and whose binding has to be performed before the detection starts (as for many pattern/antipattern categories).

In this paper we introduce an approach that aims at overcoming this limitation. We work in a fuzzy context where threshold values cannot be determined, but only their lower and upper bounds do. On this basis, the detection task produces a list of performance antipatterns along with their probabilities to occur in the model. Several refactoring alternatives can be available to remove each performance antipattern. Our approach associates an estimate of how effective each alternative can be in terms of performance benefits. We demonstrate that the joint analysis of antipattern probability and refactoring benefits drives the designers to identify the alternatives that heavily improve the software performance.

Keywords: Software Performance, Model Refactoring, Performance Antipatterns.

1 Introduction

In the software development domain, there is a high interest in the early validation of performance requirements because it avoids late and expensive fixes to consolidated software artifacts. Model-based approaches, pioneered under the name of Software Performance Engineering (SPE) by Smith [1], aim at producing performance models early in the development cycle and using quantitative

* This work has been partially supported by the European Office of Aerospace Research and Development (EOARD), Grant/Cooperative Agreement (Award no. FA8655-11-1-3055).

A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 149–164, 2015.
DOI: 10.1007/978-3-662-46675-9_10

results from model solutions to refactor the design with the purpose of meeting performance requirements [2].

Nevertheless, the problem of interpreting the performance analysis results is still quite critical. A large gap in fact exists between the representation of performance analysis results and the feedback expected by software designers. The former usually contains numbers (e.g., mean response time, throughput variance, etc.), whereas the latter should embed design alternatives useful to overcome performance problems (e.g., split a software component in two components and re-deploy one of them). The results interpretation is today exclusively based on the analysts' experience and therefore it suffers from lack of automation.

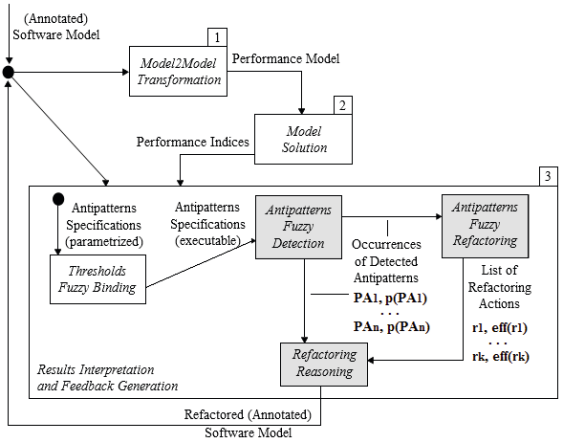

Fig. 1. Model-based software performance refactoring process

Figure 1 illustrates a model-based software performance refactoring process. It includes three main operational steps: (1) the *Model2Model Transformation* step takes as input an annotated[1] software model and generates a performance model [3]; (2) the *Model Solution* step takes as input a performance model and produces a set of performance indices [4]; (3) the *Results Interpretation and Feedback Generation* macro step takes as input both the software model and the performance indices to detect possible performance problems, and it provides a refactored (annotated) software model where problems have been removed. In particular, the refactored model is obtained with a semantics-preserving transformation that aims at improving the quality of the original software model. In other words, the functional aspects of this latter model have to remain unaltered after the transformation. For example, the interaction between two components might be refactored to improve performance by sending fewer messages with larger data per message.

[1] Software model annotations support the performance analysis by specifying parameters like workload, resource demands, etc.

A number of approaches have been recently introduced for this macro step [5, 6] (see more details in Section 2), while we were working on the detection and refactoring of *technology-independent* performance antipatterns [7–10]. *Performance antipatterns* [11] are well-known bad design practices that lead to software products suffering from poor performance. A specific characteristic of performance antipatterns is that they contain numerical parameters that represent thresholds referring to either performance indices (e.g., *high*, *low* device utilization) or design features (e.g., *many* interface operations, *excessive* message traffic).

Both the detection and solution of performance antipatterns are heavily affected by multiplicity and estimation accuracy of thresholds that an antipattern requires. For this reason, in our previous work we have experimented the influence of thresholds with respect to these two activities. We have conducted a sensitivity analysis on a case study by varying the numerical values of several thresholds for different antipatterns. Then, we have quantified threshold variations with the support of recall and precision metrics, and derived useful findings for dealing with performance antipatterns on software models [12].

The motivation of this paper stems from two main reasons:

(i) due to the stochastic nature of the process of Figure 1, it would not be realistic to assume that threshold values can be exactly determined;
(ii) it is difficult to identify refactoring actions that quicken the convergence of the whole process of Figure 1.

In this paper we introduce an approach that aims at overcoming these limitations, by providing a *thresholds fuzzy binding* that does not assign exact values to antipattern thresholds, but it works on their lower and upper bounds to make fuzzy the context of antipatterns detection and refactoring.

In such context we envisage (see Figure 1): (i) a detection task that produces a list of performance antipattern occurrences $PA_1, \ldots, PA_n$, along with their probabilities to occur, i.e. $p(PA_1), \ldots, p(PA_n)$, and (ii) a refactoring task that produces a list of available refactorings $r_1, \ldots, r_k$ with their effectiveness, i.e. $eff(r_1), \ldots, eff(r_k)$, in terms of expected performance benefits.

The contribution of this paper is to introduce:

1. A method for associating to each detected performance antipattern PA_i the probability of occurring, i.e. $p(PA_i)$.
2. A technique that estimates the *effectiveness* of each available refactoring action r_j, i.e. $eff(r_j)$, in terms of expected performance benefits.
3. A *Refactoring Reasoning* step that jointly analyzes the antipattern probability and refactoring benefits to drive the designers towards the identification of refactoring actions that quicken the process convergence. The output is a design alternative, i.e. a new (refactored) software model that undergoes the same process[2].

[2] Note that we intend to provide here an instrument to help the process convergence, hence a performance analyzer can decide to limitedly use this instrument, for example by stopping the detection and the refactoring steps before all antipatterns and refactoring actions have been devised (for sake of processing time), and then reasoning on a reduced set of alternatives.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes our approach to deal with the fuzzy antipattern detection and refactoring. Section 4 illustrates the application of our approach to a case study, i.e. an e-commerce system. Finally, Section 5 concludes the paper by outlining the most challenging research topics in this area.

2 Related Work

In literature there are some approaches that deal with the problem of improving the performance of software systems based on analysis results.

Xu et al. [5] present a semi-automated approach to find configuration and design improvement on the model level. Based on a Layered Queueing Network (LQN) model, two types of performance problems are identified, i.e, bottleneck resources and long paths. Then, rules containing performance knowledge are applied to solve the detected problems. However, the approach is notation-specific, in fact it is based on LQN rules, and it does not incorporate heuristics to rank the solutions, as suggested in this paper.

Parsons et al. [13] 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. However, the limitation of this approach is that it cannot be used in early development stages, running EJB systems are required for the detection of performance problems.

Diaz Pace et al. [14] present the ArchE framework that assists the software architect during the design to create architectures that meet quality requirements. However, defined rules are limited to improve modifiability only. A simple performance model is used to predict performance metrics for the new system with improved modifiability.

In previous work [15] we proposed an approach for automated feedback generation from software performance analysis results, based on model-driven techniques. Support to rank and solve antipatterns has been provided in [16] however thresholds have been estimated with heuristics. As future work we plan to compare our guilt-based approach with the one presented in this paper to further investigate pros and cons of the two approaches.

3 Probability-Effectiveness Approach

In this section we describe our approach by providing details on the shaded boxes of Figure 1 that represent the focus of this paper.

3.1 Antipatterns Fuzzy Detection

Performance antipatterns have been originally defined in natural language [11]. Hence, we first tackled the problem of providing a more formal representation by

introducing first-order logic rules that express a set of system properties under which an antipattern occurs [17].

As stated in Section 1, performance antipatterns are very complex (as compared to other software patterns) because they are founded on design characteristics (e.g., *many* usage dependencies, *excessive* message traffic) and performance results (e.g., *high*, *low* utilization), hence thresholds must be introduced. For example, a Blob occurs when a component requires a *lot* of information from other ones, it generates *excessive* message traffic that lead to *over utilize* the device on which it is deployed or the network involved in the communication. The logic-based formula of the Blob antipattern has been defined in [17] and reported in Equation (1), where $\mathrm{sw}\mathbb{E}$ and $\mathbb{S}$ represent the set of all software components and services, respectively.

$$\begin{aligned} &\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid \\ &(F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \\ &\vee F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects}) \\ &\wedge F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \\ &\wedge (F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \\ &\vee F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil}) \end{aligned} \tag{1}$$

In our previous work [17] we defined some heuristics to estimate thresholds numerical values. For example, the $Th_{maxConnects}$ threshold (that represents the maximum bound for the number of usage relationships a software component is involved in) has been estimated as the average number of usage relationships, with reference to the entire set of software components in the software system, plus the corresponding variance. In [12] we considered ranges of values around these average values, but the main issue was to set a suitable width to capture the actual bad practices.

To overcome such problem, in this paper we move a step ahead by determining thresholds' lower and upper bounds, thus to make fuzzy the detection step. In fact, the logic-based representation of antipatterns allows us to move the detection process in a fuzzy context by defining the probabilities associated to the logical predicates involved in antipatterns specifications. Thresholds' lower and upper bounds can be defined by examining the whole system and calculating the minimum and maximum values of the observed properties. For example, Table 1 reports the description of thresholds included in the Blob specification along with strategies to derive lower and upper bounds.

In order to calculate the probability of occurrence for each antipattern, we firstly consider the probabilities associated to logical basic predicates separately, and then we properly combine them following their logical operators, i.e. AND ($\wedge$), OR ($\vee$).

Each logical basic predicate is associated to a probability value of occurrence based on its *distance* from lower and upper bounds, respectively. In particular, we consider how far a specific design characteristic or a performance index is from the thresholds' upper bound (in case of maximum boundaries) or thresholds' lower bound (in case of minimum boundaries). This quantity is normalized

Table 1. Thresholds specification for the Blob antipattern

	Threshold	Description	Lower Bound	Upper Bound
Design	$Th_{maxConnects}$	Maximum bound for the number of connections in which a component is involved	$LBTh_{maxConnects}$ is the minimum number of connections among all the components	$UBTh_{maxConnects}$ is the maximum number of connections among all the components
	$Th_{maxMsgs}$	Maximum bound for the number of messages sent by a component in a service	$LBTh_{maxMsgs}$ is the minimum number of messages sent among all the components	$UBTh_{maxMsgs}$ is the maximum number of messages sent among all the components
Performance	$Th_{maxHwUtil}$	Maximum bound for the hardware device utilization	$LBTh_{maxHwUtil}$ is the minimum hardware utilization among all the devices	$UBTh_{maxHwUtil}$ is the maximum hardware utilization among all the devices
	$Th_{maxNetUtil}$	Maximum bound for the network link utilization	$LBTh_{maxNetUtil}$ is the minimum utilization among all the network links	$UBTh_{maxNetUtil}$ is the maximum utilization among all the network links

over the difference between thresholds' upper and lower bounds. For example, for the Blob antipattern the number of connections of a certain swE_x software component ($F_{numClientConnects}(swE_x)$) is compared to the corresponding threshold upper bound ($UBTh_{maxConnects}$) and lower bound ($LBTh_{maxConnects}$). The probability formula for this basic predicate is reported in Equation (2):

$$1 - ((UBTh_{maxConnects} - F_{numClientConnects}(swE_x))/ \\ (UBTh_{maxConnects} - LBTh_{maxConnects})) \tag{2}$$

We combine these probabilities to obtain the probability of an antipattern occurrence, following the classic probability theory formulas for independent events, that are as follows for the union of two events, (i.e., the probability of (A $\vee$ B) in the logical formula):

$$P(A \cup B) = P(A) + P(B) \tag{3}$$

and as follows for the intersection of two events (i.e., the probability of (A $\wedge$ B) in the logical formula):

$$P(A \cap B) = P(A) * P(B) \tag{4}$$

The experimentation shows that the hypothesis of independency does not compromise the validity of the approach.

3.2 Antipatterns Fuzzy Refactoring

In this section we provide an answer to a significant research question that arises when choosing a refactoring to apply on the model: "How to quantify the *effectiveness* of a refactoring action?".

To answer this question, we first observe that in case of systems representable as separable Queueing Networks (this is our case), it is well-known that the more the load of the system is balanced among nodes, the better the response time and throughput are [4, Ch. 5]. This means that, among all the available

refactorings, we should prefer to apply the one(s) resulting in a more balanced system. Hence, we need a way to quantify the *equilibrium point* of a system configuration in terms of node utilization, because utilization is the index that more directly relate to the load distribution in the system. Given the utilizations of the system's nodes in a configuration, we consider the mean utilization value as the equilibrium point.

It is evident that, the closer the utilizations of all nodes are to the mean value, the closer that configuration is to the equilibrium point. Hence, given the utilization of each node for a configuration (namely *config*), we can estimate the distance from the equilibrium point by considering the *variance* among the nodes utilizations, denoted with $var(config)$.

In order to compute $var(config)$ we need a technique to estimate the node utilizations $\bar{U}' = \{u'(n_1), u'(n_2), ..., u'(n_k)\}$ after the application of a refactoring r. In this paper we consider the following refactorings because they represent foundational actions, and it is possible to build many complex refactorings from their combination:

- *redeploy*(*Component* c, *Node* n): this action moves a software component c to the node n. Such refactoring action is aimed at improving the utilization of the node where the component c was deployed. In a performance model, this results in moving the resource demand of c to n.
- *split*(*Component* c, *Node* n): this action split a software component c in two new components c'_1 and c'_2, by properly distributing the connections of c between them, while taking into account the functional responsibilities of c. c'_1 remains on the node where c was deployed, whereas c'_2 is deployed on n. Such refactoring action is aimed at reducing the number of connections of c in an efficient way. In a performance model, this results in moving part of the resource demand of c to n.

To estimate $\bar{U}'$, we define the *unitary cost of a resource demand on a node* n, namely $uc(n)$, as follows:

$$uc(n) = \frac{u(n)}{\sum_{c \in D(n)} rd(c)} \tag{5}$$

where $rd(c)$ is the resource demand of a component c and $D(n)$ is the set of all components c that are currently deployed on n.

Hence, given two distinct nodes n_i and n_j, with $uc(n_i)$ and $uc(n_j)$ respectively, if we are moving $rd(c)$ from n_i to n_j, then

$$\bar{U}' \;=\; \{u'(n_i), u'(n_j)\} \bigcup \bar{U}_k \tag{6}$$

where

- $u'(n_i) = u(n_i) - rd(c) * uc(n_i)$
- $u'(n_j) = u(n_j) + rd(c) * uc(n_j)$
- $\bar{U}_k = \bigcup_{k \neq \{i,j\}} \{u(n_k)\}$

It is worth to notice that an algebraic approximation of future utilizations due to refactoring actions cannot be easily extended to nodes not directly involved in the split/redeploy refactorings, although they could be affected by the propagation of refactoring effects. Hence, we here introduce the assumption that utilizations of these nodes (other than $u'(n_i)$ and $u'(n_j)$) do not change after refactoring.

The above definitions based on resource demands can be straightforwardly applied to the case of mass storage resources and, in general, to any resource that in a queueing network model can be represented as a service center with queue. However, for sake of simplification, we only refer to CPU resources in our case study. For other types of resources, such as RAMs, the concepts of utilization and resource demands are not applicable as they are, so these definitions do not hold. Hence, for all cases where resources like RAM could be critical, a different quantification of performance improvements due to refactoring has to be introduced.

After estimating utilizations of each possible refactored configuration $config'$, we can estimate $var(config')$.

At this point, with respect to the research question that we have arisen above, we intend to distinguish if a refactoring r_i is *better* than a refactoring r_j. For this goal, denoting by $config'_i$ and $config'_j$ the configurations resulting from r_i and r_j respectively, if $var(config'_i) < var(config'_j)$ then we assess r_i as better than r_j. This is because r_i results in a more balanced refactored configuration (i.e. one that is closer to the equilibrium point) than r_j.

Now, starting from the performance shown by an initial model, we make a distinction between *beneficial* and *non-beneficial* refactorings. In particular, a refactoring r is *beneficial* if it results in a refactored configuration $config'$ that shows better performance than the ones shown by the initial configuration $config_{Initial}$. Conversely, r is *non-beneficial* if performance do not improve by applying it.

With this distinction in mind, we define the *effectiveness* of a refactoring r as $eff(r) = var(config_{Initial}) - var(config')$. Basing on $eff(r)$, we can now classify the set of available refactorings. In particular, we consider r as a *beneficial* refactoring if $eff(r) > 0$, otherwise as *non-beneficial*. In practice, the $eff(r)$ value suggests the degree of influence that r can provide to the performance of the resulting configuration, i.e. $config'$.

3.3 Refactoring Reasoning

As shown in Figure 1, the refactoring reasoning operational step takes as input: (i) the list of detected antipatterns associated with their probability of occurrence; (ii) the list of refactoring actions associated with their effectiveness. Several strategies can be devised to make use of this knowledge, for example:

- *High probability*: while looking at the list of detected antipatterns it is possible to identify the ones that most likely represent a bad practice, and then to apply (one of) the most effective refactorings to (one of) them;

- *High effectiveness*: while looking at the list of refactoring actions it is possible to identify the ones that most likely provide a performance improvement, and then to apply (one of) the most effective refactorings to (one of) the most likely occurring antipatterns;
- *High combination of probability and effectiveness*: while looking at the list of detected antipatterns and refactoring actions it is possible to combine the $(probability, effectiveness)$ values to identify the ones that most likely provide a performance improvement while removing bad practices.

The goal of providing probability and effectiveness values for antipatterns is to introduce an ordering in the list of detected antipatterns, where highly ranked antipatterns are the most promising causes for performance problems as well as the most promising candidates to solve such problems. The key factor of our approach is to consider the thresholds' lower and upper bounds thus to evaluate the whole system. We first assign a probability to each antipattern, and then we estimate the effectiveness of its refactoring actions on the basis of the achieved equilibrium point.

4 Case Study: E-Commerce System (ECS)

E-Commerce System (ECS) is a web-based system that manages business data related to books and movies. A *Guest* may invoke the *BrowseCatalog* service, whereas a *Customer* may invoke two services, i.e., *Login* and *MakePurchase*. Several software components and hardware platforms have been defined in ECS. For the sake of simplicity we name components $C1, \ldots, C6$ and platforms $n1, \ldots, n3$, in particular $C1$ and $C2$ are deployed on $n1$, $C3$ and $C4$ are deployed on $n2$, and $C5$ and $C6$ are deployed on $n3$.

Among all system services, we focus here on the *MakePurchase*, which is triggered whenever a customer wants to purchase a book or a movie, after authentication. We assume that a performance requirement have been defined on the *MakePurchase* service, i.e. its average response time must not exceed 90 seconds. Such requirement must be fulfilled under a workload of 100 customers. The performance analysis has been conducted by transforming the software model into a Queueing Network (QN) model [18] and by solving the latter with the Java Modeling Tools (JMT) [19].

The considered requirement is violated because, under a workload of 100 users purchasing a product, the mean time elapsed in the server-side for each request (i.e., the average response time at the server-side) is 93.79 seconds that is larger than the stated requirement.

4.1 ECS: Antipatterns Fuzzy Detection

As stated in Section 3.1, antipatterns fuzzy detection is performed by assigning a probability value for the logic basic predicates. In the following we illustrate our approach applied to the Blob occurrences of the ECS case study, however

other antipatterns (i.e., the Concurrent Processing System and the Pipe and Filter [11]) have been detected and analyzed.

Table 2 reports the observed values of the ECS case study. In particular the first column reports all the software components listed as candidates for the Blob occurrence. The remaining columns reports the values coming from the application of functions defined in the Blob logic-based specification (see Equation 1). For example, in the first row of Table 2 we can notice that the $C1$ component has 3 connections, it sends 11 messages, and it is deployed on an hardware platform with an utilization of 57%. In the lower part of Table 2 we report thresholds' upper and lower bounds that have been calculated across all the system features.

Table 2. EHS: Thresholds upper and lower bounds for the Blob antipattern

Component	$F_{numClientConnects}(swE_x)$	$F_{numMsgs}(swE_x, swE_y, S)$	$F_{maxHwUtil}(P_{xy}, all)$
C1	3	11	0.57
C2	5	16	0.57
C3	7	25	0.87
C4	6	22	0.87
C5	1	4	0.41
C6	3	13	0.41
Upper Bound	7	25	0.87
Lower Bound	1	4	0.41

Table 3. EHS: Fuzzy detection for the Blob antipattern

Component	p(A)	p(B)	p(C)	p(Blob)
C1	0.33	0.33	0.34	0.04
C2	0.67	0.57	0.34	0.13
C3	1	1	1	1
C4	0.83	0.86	1	0.71
C5	0	0	0	0
C6	0.33	0.43	0	0

Table 3 reports the values calculated according to Equations (4) (3), and it is structured as follows. The first column report the set of all the software components. The subsequent three columns report the probabilities of the sub equations (namely A, B, C) included in the Blob specification. For Blob occurrence the event A is associated to the sub equation related to the number of connections, and it is calculated using the formula reported in Equation (2). For example, Table 3 shows that the $C1$ component is associated with p(A) calculated as: 1- (7-3/7-1) = 1 - 4/6 = 0.33. The last column reports the p(Blob) that represents the probability of occurrence for the corresponding components. For example, Table 3 shows that the $C1$ component is associated to a probability equal to 0.04 (= 0.33 * 0.33 * 0.34) that basically represents the confidence associated in the detection of the $C1$ component as a Blob occurrence.

Similarly to Table 3 fuzzy detection of antipatterns have been performed to estimate the Concurrent Processing Systems (CPS) and Pipe & Filter occurrences. Numerical values are reported in Section 4.3.

4.2 ECS: Antipatterns Fuzzy Refactoring

Figure 2 shows the classification of the refactored configurations vs the initial one: on the x-axis the refactoring efficiencies ($eff(r)$) are reported, whereas the y-axis represents the response time (RT, calculated by simulating the underlying performance model) of the corresponding refactored configuration.

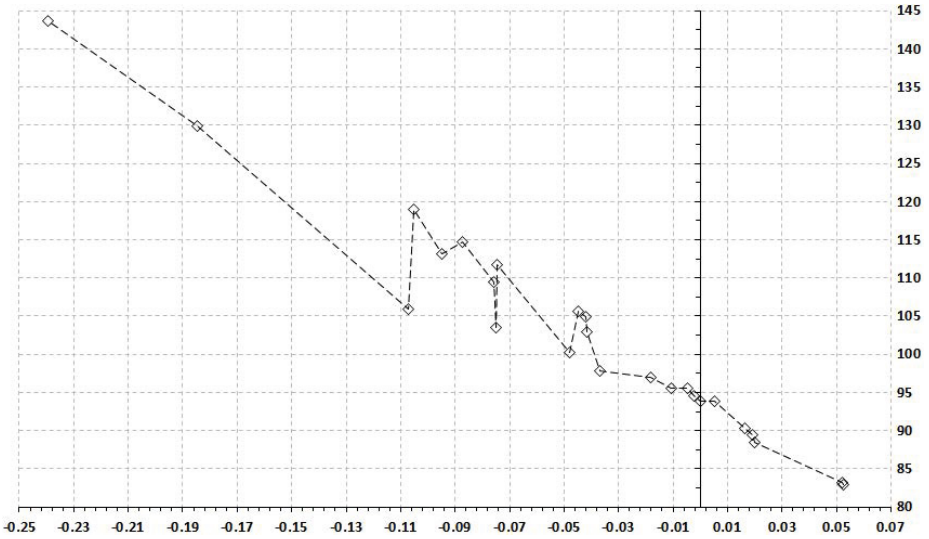

Fig. 2. Refactoring effectiveness vs. system response time

Our approach bases on the conjecture that we mentioned in Section 3.2, i.e. "we consider r as a *beneficial* refactoring if $eff(r) > 0$, otherwise as *non-beneficial*.". Hence, basing on our approach, there are 6 beneficial actions, whereas all the remaining ones are non-beneficial. To validate this, we simulated each refactored configuration, obtaining its response time and comparing it to the one of the initial configuration (i.e. the point with $x = 0$ in Figure 2). Given this knowledge, we computed *recall* and *precision* of our approach on the ECS case study[3]. In particular, the recall is 100%, because all the 5 actual beneficial refactored configurations (i.e. the 5 right-most points) have been retrieved; instead, the precision is 83.33%, because among the 6 refactored configurations that we computed (i.e. the 6 points with $x > 0$), 5 were actually beneficial.

The conjecture above is true for all the points of Figure 2, except for the point $(0.005, 93.79)$, i.e. the first point in the positive side of x-axis. This is due to the fact that the corresponding refactoring $Split(C1, n3)$ is a *border-line case*; in fact, by simulating the resulting refactored configuration several times,

[3] Precision and recall are well-known metrics aimed at quantifying the effectiveness of a technique for pattern recognition or information retrieval [20]. High recall means that the technique has returned most of the relevant results, while high precision means that it has returned substantially more relevant results than irrelevant ones.

some times the response time deteriorates, other times it improves. Note that, in this experimentation, we simulated each refactored configuration just once. As a future work, it would be interesting to take into account this "pathological uncertainty", by simulating N times each refactored configuration, and counting how many times it actually results in a better configuration with respect to the initial one. Thus, the existence of borderline cases influences precision. In fact, if at least one border-line case exists, then the precision is strictly less than 100%. Our conjecture still holds up to borderline cases. However, since we are interested in directing the user to obtain the best benefit (if one exists), by choosing the action with the maximum effectiveness we can be confident that if that action is beneficial, then it is the most beneficial one.

Observing Figure 2, we can notice that there are some "classification errors". Given a configuration, it is not well-classified if there exists at least a different configuration having greater effectiveness (thus, it is considered better by our approach) and lower RT (thus, actually it is not better) at the same time. Our approach made 6 classification errors on the case study: one error with respect to beneficial refactorings and 5 errors with respect to non-beneficial ones. The former can be discarded, because it is due to the border-line case. Concerning the latter errors, we notice that the probability of making a classification error increases while effectiveness decreases; in fact, for very small values of $eff(r)$, RT is very high.

4.3 ECS: Refactoring Reasoning

In our case study, the refactoring reasoning is based on the occurrences of three different performance antipatterns, i.e., *Blob*, *CPS* and *P&F*. Figure 3 shows the summary of our experimentation coming from the antipatterns fuzzy detection and refactoring steps of the model-based software refactoring process (see Figure 1): each (x, y) point is related to a considered antipattern and a refactoring action, where x is the antipattern occurrence probability, and y is the refactoring effectiveness. We are obviously interested to upper right-most points.

Figure 3 shows that if we use the *High probability* strategy (see Section 3.3) for the Blob antipattern, occurrences associated with a high probability are actually the ones that most likely provide a performance benefit.

For example, concerning the Blob antipattern: (i) points $(1, y)$ refer to component $C3$ that has a probability of occurrence equal to one, and two of its refactoring actions bring a system performance improvement, e.g. one is $Split(C3, n3)$ as labeled in the figure; (ii) points $(0.71, y)$ refer to component $C4$ that has a probability of occurrence equal to 0.71, and three of its refactoring actions have been experimented to be beneficial for the system, e.g. one is $Split(C4, n3)$ as labeled in the figure; (iii) all the remaining points correspond to low or zero probabilities of occurrence and refer to components ($C1$, $C2$, $C5$, $C6$) for which refactoring effectiveness is very low.

Interestingly, in Figure 3 we can notice that CPS antipattern occurrences associated with a high probability may not imply a performance improvement. In our ECS case study we found that in case of the ($n2$, $n3$) CPS occurrence,

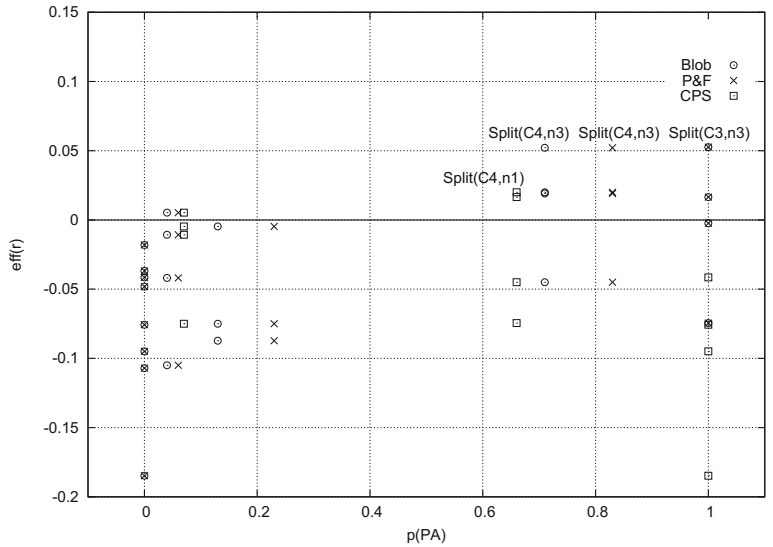

Fig. 3. Probability of antipattern occurrence vs. refactoring effectiveness

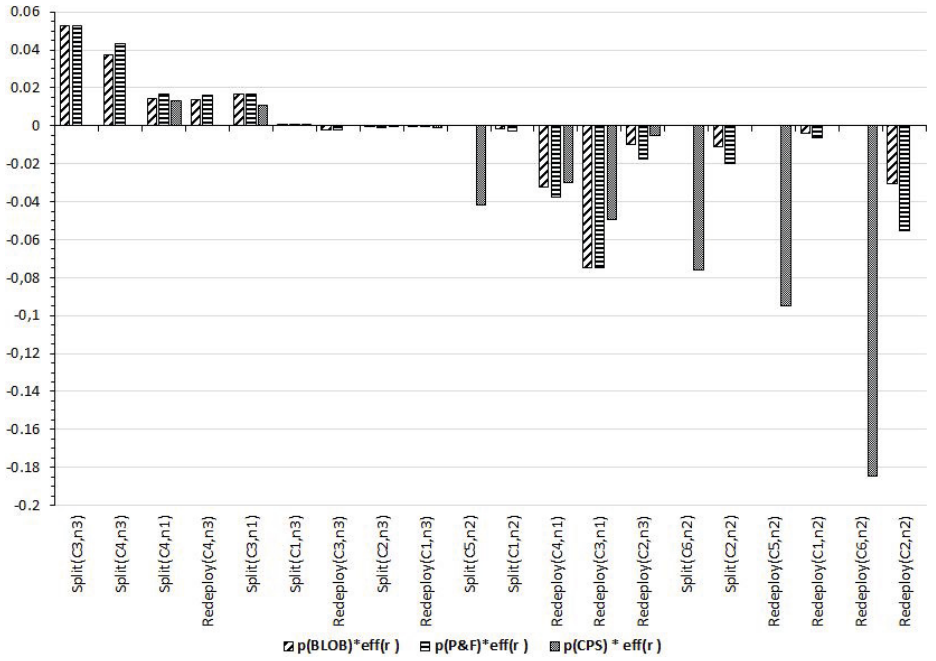

Fig. 4. Refactoring vs. probability of antipattern occurrence multiplied by refactoring effectiveness

which has a probability of one, no refactoring actions are actually beneficial for the system performance.

Figure 3 also shows that if we use the *High effectiveness* strategy (see Section 3.3) for the Pipe & Filter antipattern, refactoring actions associated with a high effectiveness, e.g. $Split(C4, n3)$ as labeled in the figure, are actually the ones that most likely remove a bad practice.

It is worth to notice that component $C3$ has a probability of occurrence equal to one also as a Pipe & Filter, besides as a Blob. This generates the problem of duplicated antipatterns since the same model element may represent two different antipattern occurrences. On the contrary, component $C4$ has a quite high probability of occurrence as a Pipe & Filter (i.e., 0.83) and it is larger than the probability found as a Blob occurrence (i.e., 0.71). All the remaining components ($C1$, $C2$, $C5$, $C6$) have a low or zero probability of occurrence and their effectiveness is in fact very low.

The experimentation highlighted that our approach is able to provide some guidelines to software designers in the selection of refactoring actions. In particular, we found that antipatterns with a low probability of occurrence do not entail any performance effectiveness. On the contrary, antipatterns with a high probability of occurrence may include beneficial refactoring actions but their effectiveness is not guaranteed in advance.

Figure 4 shows the product between probability of antipattern occurrence and refactoring effectiveness, to support the *High combination of probability and effectiveness* refactoring strategy (see Section 3.3). In this case antipatterns showing different probabilities of occurrence may result similar in their combination with the effectiveness of available refactorings. For example, the evaluation of $p(PA) * eff(r)$ for the P&F antipattern points out that even if $C3$ and $C4$ have a probability of occurrence equal to 1 and 0.83, respectively, then their combination values are quite similar while considering the refactoring actions $Split(C3, n1)$ and $Split(C4, n1)$, in fact they result in 0.0165 and 0.0166, respectively.

5 Conclusion

In this paper we have presented an approach for performance-based software model refactoring. The novelty of the approach is that it works in a fuzzy context where: (i) the detection of antipatterns additionally produces their probabilities to occur in the model; (ii) the refactoring of antipatterns additionally indicates the effectiveness of design alternatives in terms of performance benefits. Our case study have demonstrated that the joint analysis of antipattern probability and refactoring effectiveness drives the designers to identify the alternatives that heavily improve the software performance.

This work is embedded in a wider research area that is the interpretation of performance analysis results and the generation of feedback. A lot of work has to be done to validate and refine the presented methodology. For example, as future work, we plan to integrate our approach with the other work that we have conducted up today in this area, particularly with respect to [16]. Moreover, we are facing the problem of using the effectiveness of refactoring actions

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: (i) the consideration of multiple resources at the same time (e.g. a split can relieve the CPU load while aggravating the network occupancy due to increased interactions), (ii) the simultaneous application of multiple refactoring actions.

References

1. Smith, C.U.: Introduction to software performance engineering: Origins and outstanding problems. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 395–428. Springer, Heidelberg (2007)
2. Woodside, C.M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. In: Briand, L.C., Wolf, A.L. (eds.) FOSE, pp. 171–187 (2007)
3. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis, pp. 1–190. Springer (2011)
4. Lazowska, E., Kahorjan, J., Graham, G.S., Sevcik, K.: Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice-Hall, Inc. (1984)
5. Xu, J.: Rule-based automatic software performance diagnosis and improvement. Perform. Eval. 69, 525–550 (2012)
6. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: ICPE, pp. 105–116 (2010)
7. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Digging into UML models to remove performance antipatterns. In: ICSE Workshop Quovadis, pp. 9–16 (2010)
8. Trubiani, C., Koziolek, A.: Detection and solution of software performance antipatterns in palladio architectural models. In: International Conference on Performance Engineering (ICPE), pp. 19–30 (2011)
9. Arcelli, D., Cortellessa, V., Trubiani, C.: Antipattern-based model refactoring for software performance improvement. In: ACM SIGSOFT International Conference on Quality of Software Architectures (QoSA), pp. 33–42 (2012)
10. Cortellessa, V., De Sanctis, M., Di Marco, A., Trubiani, C.: Enabling Performance Antipatterns to arise from an ADL-based Software Architecture. In: Joint Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA (2012)
11. Smith, C.U., Williams, L.G.: More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In: International Computer Measurement Group Conference, pp. 717–725 (2003)
12. Arcelli, D., Cortellessa, V., Trubiani, C.: Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns. ECEASST 59 (2013)
13. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Journal of Object Technology 7, 55–91 (2008)
14. Diaz-Pace, A., Kim, H., Bass, L., Bianco, P., Bachmann, F.: Integrating quality-attribute reasoning frameworks in the arche design assistant. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 171–188. Springer, Heidelberg (2008)

15. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Approaching the Model-Driven Generation of Feedback to Remove Software Performance Flaws. In: EUROMICRO-SEAA, pp. 162–169. IEEE Computer Society (2009)
16. Trubiani, C., Koziolek, A., Cortellessa, V., Reussner, R.: Guilt-based handling of software performance antipatterns in palladio architectural models. Journal of Systems and Software 95, 141–165 (2014)
17. Cortellessa, V., Di Marco, A., Trubiani, C.: An approach for modeling and detecting software performance antipatterns based on first-order logics. Software and System Modeling 13, 391–432 (2014)
18. Cortellessa, V., Mirandola, R.: PRIMA-UML: a performance validation incremental methodology on early UML diagrams. Sci. Comput. Program. 44, 101–129 (2002)
19. Casale, G., Serazzi, G.: Quantitative system evaluation with Java modeling tools. In: International Conference Performance Engineering, pp. 449–454. ACM (2011)
20. Frakes, W.B., Baeza-Yates, R.: Information retrieval: data structures and algorithms. Prentice-Hall, Inc., Upper Saddle River (1992)