Automating Performance Antipattern Detection and Software Refactoring in UML Models

Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo
Department of Computer Science and Engineering, and Mathematics
University of L'Aquila (Italy)
{davide.arcelli,vittorio.cortellessa,daniele.dipompeo}@univaq.it

***Abstract*—The satisfaction of ever more stringent performance requirements is one of the main reasons for software evolution. However, it is complex to determine the primary causes of performance degradation, because they may depend on the joint combination of multiple factors (e.g., workload, software deployment, hardware utilization). With the increasing complexity of software systems, classical bottleneck analysis shows limitations in capturing complex performance problems. Hence, in the last decade, the detection of performance antipatterns has gained momentum as an effective way to identify performance degradation causes. We introduce PADRE (Performance Antipattern Detection and REfactoring), that is a tool for: (i) detecting performance antipattern in UML models, and (ii) refactoring models with the aim of removing the detected antipatterns. PADRE has been implemented within Epsilon, an open-source platform for model-driven engineering. It is based on a methodology that allows performance antipattern detection and refactoring within the same implementation context.**

***Index Terms*—Software Performance, Model-Driven Development**

I. Introduction

Performance-driven software evolution presents a high complexity due to two causes: (i) performance is a quality attribute that emerges from the combination of different factors (e.g., workload, software deployment, hardware utilization), hence it is difficult to estimate in advance the impact of software changes on such attribute; (ii) the causes of performance degradation are ever more difficult to detect, due to the complexity and heterogeneity of software systems.

The analysis of performance indices has been traditionally based on bottleneck identification and removal, which has been applied for decades on traditional software systems. More recently, performance antipatterns have emerged as more effective instruments to detect and remove non-trivial performance problems both on models [18] and on code [14]. A performance antipattern is a formalization of a bad design practice that induces performance degradation [17], and it consists of conditions for its detection along with refactorings for its removal.

Many critical performance problems usually emerge in the late lifecycle phases, because system integration highlights the combination of factors mentioned above. However, the identification and removal of performance problems in the initial lifecycle phases, conducted on early artifacts as models, allows to narrow down the search space for possible later empirical performance tuning.

This research was supported by the Electronic Component Systems for European Leadership Joint Undertaking through the MegaMart2 project (grant agreement No 737494).

In this paper, we present PADRE (Performance Antipattern Detection and REfactoring), a tool for detection and removal of performance antipatterns on UML models. PADRE is founded on a well-assessed approach for the formalization of performance antipattern conditions with first-order logics [6], and on a methodology for model refactoring aimed at removing performance antipatterns [2]. It has been designed and implemented within Epsilon [9], that is a widely adopted Eclipse-based open-source model-based software engineering platform.

A novelty of PADRE is that it introduces automation in the performance antipattern detection, in the identification and execution of refactorings that can remove antipatterns. PADRE can be currently used to suggest possible architectural refactoring that might show better performance than the current system architecture. In perspective, it can be adopted in production, by exploiting traceability links between a running system and its corresponding model, which could be fed with actual performance measurements coming from profiling results; once a possible beneficial evolution is identified, the corresponding refactoring should be (semi-)automatically propagated to the implementation level.

The rest of this paper is organized as follows: Sec. II discussed related work; Sec. III provides tool-related explanations of PADRE goals, requirements, architecture and inner workings. Sec. IV shows PADRE at work by means of an illustrative example and Sec. V concludes the paper and highlights the main future research goals that we intend to pursue.

II. Related Work

In the last decade, models have been increasingly used from the early phases of software development down to the evolution phase, where software evolves (usually through refactoring steps) for many reasons, like new requirements, changes of context, etc. In practice, the application of refactoring to models helps to analyze and compare suitable architectural alternatives before applying them on the real system.

Most of recent papers in this area deal with model-based refactoring driven by functional properties [3], [7], [8], [12], whereas only few of them consider non-functional properties.

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

As reported in a recent systematic literature review (SLR) on UML model refactoring [13], some approaches use multi-views for conducting model refactoring. Most of them only work exclusively on Class or Activity Diagrams, whereas none of them exploits the information in Deployment Diagram that is, instead, relevant for performance analysis. Hence, we base our approach on UML models that include Component, Sequence and Deployment Diagrams together. From the cross-checking of properties in these diagrams, along with performance indices, we detect performance problems in UML models and identify refactoring that can remove these problems.

Several approaches have been introduced in literature to use performance antipattern knowledge for detecting and removing performance problems in software models [5].

Koziolek et al. [10] presented an automated approach for performance improvement driven by architectural tactics. The approach describes a multi-objective evolutionary optimization algorithm searching optimal trade-offs in the design space, in the context of Palladio Component Models (PCMs). The main limitation of this approach is that it is time-consuming because the design space may be huge. Instead, the PADRE approach does not look for optimality but to satisfy performance requirements.

Wert et al. [20] presented an evolution of Dynamic Spotter [19], where they have introduced heuristics for measurement-based detection of five well-known performance antipatterns in inter-component communications. Differently, PADRE spans over a set of heterogeneous antipatterns (i.e., not only at inter-component communication level) and it is not limited to their detection but proposes refactoring that may remove them and may improve performance.

Xu [21] presented a prototype named Performance Booster, used in the early design phases. Performance antipatterns are detected on a Layered Queuing Network (LQN) obtained from a software model by means of a bi-directional transformation. Refactoring takes place on the performance model and the corresponding refactored software model is obtained by exploiting transformation bi-directionality. However, performance models are more abstract than software models, hence the portfolio of refactoring available on the former is much more limited than the one on the latter [1].

Parsons et al. [15] analyzed EJB performance antipatterns represented as a set of rules loaded into a detection engine. The application monitoring leads to reconstruct the run-time design and properties. The detection rules are matched on the obtained software model to identify the detected EJB antipatterns. However, this approach deals with technology-specific performance antipatterns, and the application deployment is not considered, so refactoring related to components re-deployment are not considered.

To the best of our knowledge, PADRE is the first tool that implements, within an unique environment (i.e., Epsilon), a thorough approach for antipattern detection and removal on UML models.

III. PADRE TOOL

We provide explanations of PADRE goals, requirements, architecture and underlying methodology.

A. Goals of the Tool

The main concern of PADRE is model refactoring driven by performance antipatterns detection. In its current implementation, PADRE supports UML models profiled with MARTE[1]. Such profiling represents a performance-oriented model annotation step, where performance analysis results are retrieved, processed and finally reported back on the UML model. Performance analysis consists of the resolution of an analytic performance model, generated from the UML-MARTE model, by means of Mean-Value Analysis [11].

B. Tool Requirements

There are several requirements that have to be taken into account to use PADRE:

1) The UML system model must adhere to an UML subset, which basically spans among three different views of the system: static, dynamic, and deployment view.
   - The *static view* is represented through a UML Component Diagram, which involves software Components, Interfaces and their Operations, as well as Components' Interface Realizations and Usages.
   - The *deployment view* is represented through a UML Deployment Diagram, which describes Artifacts (i.e., Component instances), Devices (i.e., platform nodes), as well as the corresponding deployment of the former onto the latter.
   - The *dynamic view* is represented through a set of UML Interactions (in particular Sequence Diagrams), one for each system Use Case, where Component instances are associated to Lifelines and Messages represent Component operation calls.
2) The performance-oriented annotation of the UML system model must adhere to a MARTE subset (coming from MARTE GQAM package), covering the dynamic and deployment views of the UML system model, as summarized in the following[2]:
   - The GaExecHost stereotype can be applied to Devices within the deployment view to define: (i) three input performance parameters, namely the scheduling policy (*schedPolicy*), the device speed (*speedFactor*), and its multiplicity (*resMult*), and (ii) one output parameter carried out by performance analysis, namely the device *utilization*.
   - The GaStep and GaWorkloadEvent stereotypes can be applied to Use Cases within the dynamic view to define three input performance parameters, namely the execution probability for the Use Case (GaStep's

[1] https://www.omg.org/omgmarte/

[2] Performance-oriented annotation may even be restricted to a sub-system model, in case only that limited part of the system is considered performance-critical and therefore of interest for antipattern detection and removal.

prob tag), its number of repetitions (GaStep's *rep* tag) and the arrival pattern for the workload represented by the Use Case (GaWorkloadEvent's *pattern* tag). In addition, GaScenario from MARTE GQAM can be applied to each Use Case to define two output performance parameters, namely the mean response time (*respT*) and the mean *throughput* of the Use Case.

- The GaAcqStep stereotype can be applied to Messages within Use Cases' Interactions to specify four input performance parameters, namely the execution probability for the operation request induced by the Message (*prob*), its number of repetitions (*rep*), possible size of the information transmitted through the Message (*msgSize*), and a service demand for the operation request (*servDemand*).

3) Currently, PADRE only uses JMVA, which is the JMT toolset[3] performance model solver based on exact Mean-Value Analysis that applies only to product-form performance models [4]. This limits the PADRE scope to performance antipatterns detectable through "snapshots" of the system, i.e. the ones that do not need to collect performance indices over time.

C. Tool Workflow

Fig. 1 describes the (iterative) methodology underlying PADRE. We denote the initial (unsatisfactory) system model by M and a (potentially satisfactory) refactored model by M_R. Blue-colored boxes represent activities within the workflow that are connected through tick arrows representing the control flow. Thin arrows represent the object (i.e., models) flow.

If the initial model M satisfies all the tool requirements, then PADRE workflow can start. The process begins with the execution of a model-to-text transformation (namely *UML-MARTE To JMVA*), which takes M as input and produces a textual .jmva file containing the specification (in XMI format) of a performance analytic model in product-form corresponding to M and solvable with JMVA[4]. After the performance analysis (see the JMVA box in Fig. 1), the obtained indices are properly propagated back to the system model, through the *Performance Analysis Results Back-Annotation* activity. The latter has to properly extract, process and report the resulting performance indices from the .jmva file to M, while adhering to the profiling requirements defined in Sec. III-B[5].

Once M has been property filled, a performance analysis interpretation and feedback generation sub-process is executed, in the form of a *Performance Antipatterns Detection* activity followed by user-driven *Antipattern-Based Model Refactoring*. Such two-steps sub-process represents the key-feature of the PADRE methodology, aimed at improving the performance shown by M for sake of performance requirements fulfilment.

[3]http://jmt.sourceforge.net/

[4]Note that, in order to produce the .jmva file, PADRE relays on a pivotal model conforming to a JMT metamodel that has been reverse-engineered from an XML Schema internally used by JMT.

[5]As for the generation of the JMVA analytic model, also the back-annotation exploits a pivotal JMT model conforming to the reverse-engineered JMT metamodel.

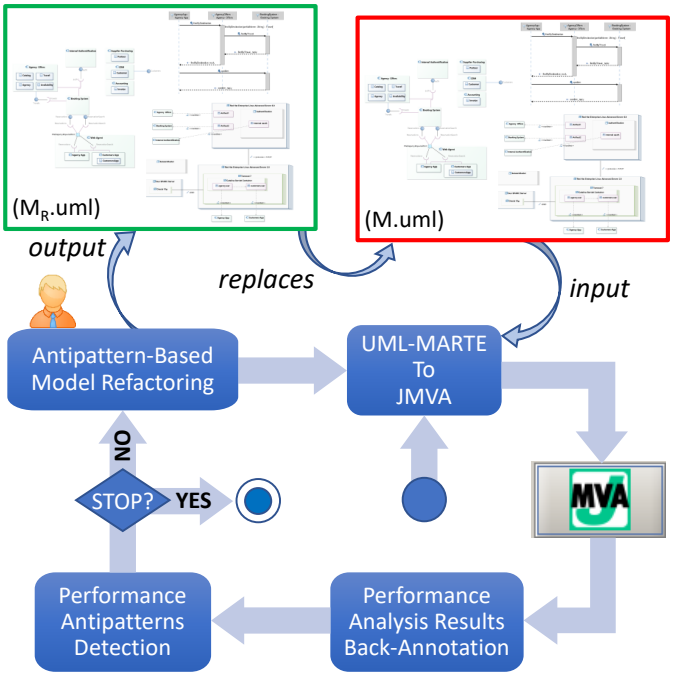

Fig. 1: PADRE workflow.

By exploiting the Epsilon platform and, in particular, the Epsilon Validation Language, PADRE provides a unifying working environment for performance analysis interpretation and feedback generation sub-process, which represents the major contribution of the tool. The EVL engine provides an infrastructure that: (i) checks properties against a model, (ii) enables a set of predefined refactorings, and (iii) allows users to execute one refactoring at a time, with the aim of satisfying the properties. Properties to check and possible refactorings are codified into .evl file(s) that can be processed by the EVL engine. Hence, PADRE grounds on a specific .evl file containing the formalization of knowledge concerning performance antipatterns, namely: (i) the antipattern detection rules that comes from their occurrence conditions, and (ii) refactorings aimed at removing detected antipatterns from the model.

At each application of a refactoring, a new system model is obtained, namely M_R, which replaces M for a new iteration of the PADRE workflow. The process ends arbitrarily by stopping the workflow execution, which could happen once a model satisfies the performance requirements.

D. Tool Architecture

Fig. 2 illustrates the current PADRE's architecture. At the rhs, PADRE macro-component is depicted, with its main inner components and interfaces implementing the workflow of Fig. III-C. Instead, on the left, external macro-components are shown (i.e., Epsilon and jmt.jmva).

To exploit the EVL execution semantics that enables the performance antipatterns detection and model refactoring in a unique environment, most of PADRE inner components specialize corresponding EVL components to tailor the typical

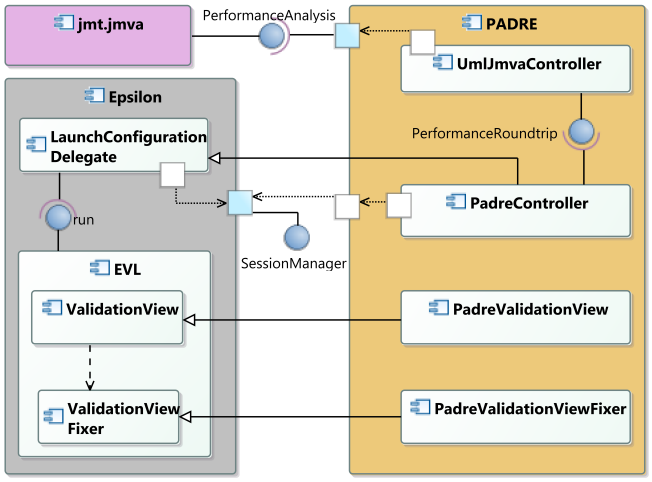

Fig. 2: PADRE architecture.

EVL workflow, for example by introducing the performance analysis roundtrip execution needed to detect antipatterns in models subject to the PADRE workflow.

PadreController is in charge of properly executing and managing the control flow of the PADRE workflow. In Fig. 2, by exploiting the port redefinition mechanism provided by the UML notation, a local PadreController's port redefines a global PADRE's port which, in turn, redefines the global Epsilon's port realizing the SesssionManager interface. In the typical Epsilon workflow, such interface is indirectly exposed by the LaunchConfigurationDelegate component and Epsilon can launch model checking and refactoring sessions. In case of EVL, LaunchConfigurationDelegate runs and interacts with the EVL ValidationView, which basically represents the output interactive console for EVL.

The performance analysis needed for antipatterns detection (see PerformanceRoundtrip interface) is provided by an ad-hoc PADRE's inner component, namely UmlJmvaController, which exploits JMT libraries for exact MVA (see PerformanceAnalysis interface provided by jmt.jmva). While redefining the SesssionManager interface realization within PADRE, the performance analysis has been properly introduced within the original EVL workflow.

In EVL, ValidationView depends on ValidationViewFixer; the former exploits the latter to enable and execute refactoring. PadreValidationView and PadreValidationViewFixer are specializations of EVL's ValidationView and ValidationViewFixer, respectively, where: EVL's ValidationView is a visual interactive console where model analysis results are listed, and it uses ValidationViewFixer.

IV. Illustrative Example

In this section, we show PADRE at work by means of an illustrative example[6].

Fig. 3(a) shows an excerpt of *Pipe and Filter* (PaF) performance antipattern definition [6], i.e. the detection rule and some of the available refactoring(s). A performance antipattern definition is an EVL `critique` that applies to a specific context (i.e., a specific UML metaclass) and that contains an antipattern detection rule (namely `checks`), a message to return when the antipattern is detected and a set of imperative blocks (namely `fix`es), each codifying a possible model refactoring.

The PaF detection rule is applied to UML Operations and consists of four EVL operations returning a boolean (namely *PaF_F_probExec()*, *PaF_F_resDemand()*, *PaF_F_throughtput()*, and *PaF_F_maxHwUtil()*), each representing a predicate of a first-order logic formula in conjunctive normal form.

The `do` block of a `fix` contains a call to an ad-hoc EVL operation (see `moveToNewComponentOnNewDevice` and `moveToNewComponentOnLessUsedNearDevice` of Fig. 3(a)), which codifies a possible model refactoring that might lead to the removal of the PaF occurrence and, hopefully, to a performance improvement[7].

The kind of support currently provided by PADRE is named *User-driven multiple refactoring* [2] and strictly depends on the execution semantics of the EVL language [9]. It consists of interactive antipattern detection and refactoring, where antipattern occurrences (i.e., *critiques*) are detected on the software model and a number of available refactorings (i.e., *fixes*) are then selected by the user. Each selection triggers the application of the refactoring on a temporary version of the model. When the user stops the refactorng session, the temporary software model is finalized. Such "freezing" of the refactoring session, which is native for the EVL execution engine, is not suitable in our context, because new performance indices are needed for a new antipattern detection. Moreover, if a new element is created by a refactoring, then the latter cannot be referred in subsequent refactorings. PadreValidationView (see Fig. 2) overrides such native EVL engine behaviour by updating each refactored temporary version of the model with performance indices carried out by performance analysis.

Fig. 3(b) shows a snapshot of PadreValidationView right after a performance antipatterns detection: all the detected occurrences are listed in the validation view and, for each occurrence, both the context metaclass (e.g., Artifact) and the metaclass instance representing the antipattern source (e.g., BooksCatalog) are listed[8]. By right-clicking a list item, available refactorings aimed at removing such antipattern occurrence are proposed, and the user can apply one of them. For example, among the 11 occurrences (i.e., 3 Blobs, 6 CPSs and 2 PaFs) [6] reported in Fig. 3(b), four refactorings are applicable to the PaF having the *login* Operation as source: the first two, namely *Move it to a new Component deployed to a new Node* and *Move it to a new Component deployed to the less used*

[6]PADRE at work is more extensively illustrated in the video at https://youtu.be/D-WcsL5reDY.

[7]For sake of space, we do not provide details on the implementation of such refactorings, as PADRE can be found at https://github.com/SEALABQualityGroup/padre/tree/towardsPerformanceLoop.

[8]Note that context metaclass helps the user to disambiguate the antipattern source, because in UML it is not forbidden to associate the same instance name to different elements in the model.

```
context Operation {
    critique PaF {
        check:
            not (self.PaF_F_probExec() and self.PaF_F_resDemand() and
                 (self.PaF_F_maxHwUtil() or self.PaF_F_throughput()))
        message : "PaF <Operation> " + self.name
        fix {
            title : "Move it to a new Component deployed to a new Node"
            do {
                self.moveToNewComponentOnNewDevice();
            }
        }
        fix {
            title : "Move it to a new Component deployed to the" +
                    " less used neighbour Node"
            do{
                self.moveToNewComponentOnLessUsedNearDevice();
            }
        }
    }
}
```

(a) Excerpt of a rule for the Pipe and Filter antipattern.

(b) Example of PADRE refactoring session.

Fig. 3: Performance antipatterns detection and model refactoring with PADRE.

neighbour Node, correspond to the `fix`es of Fig. 3(a) and, in particular, to `moveToNewComponentOnNewDevice` and `moveToNewComponentOnLessUsedNearDevice` EVL operations, respectively.

The effect of every refactoring on the performance indices is hard to predict in advance, because performance is an emerging property that depends on many factors. However, as shown in [18], support can be provided for driving the whole process towards satisfactory results.

V. Conclusion and Future Work

In this paper we have introduced PADRE, that is a tool implementing a methodology for performance antipatterns detection and refactoring in UML models profiled with MARTE.

In the current PADRE implementation, performance analysis consists in solving an analytic performance model, generated from the UML-MARTE model, by means of Mean-Value Analysis. In a more sophisticated usage scenario that we plan to support in a mid-term future (e.g., for sake of software evolution), performance indices can be extracted from a system implementation by means of monitoring and log analysis. After the refactoring phase, once a satisfactory model is obtained, the refactoring shall be propagated from the model to the implementation level. This latter task is not trivial because it needs a non-ambiguous mapping between model and code that must ensure their behavioural equivalence.

We are also working on widening the PADRE scope to more sophisticated antipatterns, based on the observation of performance indices over time, and aimed at identifying critical trends such as huge spikes or linearly increasing response times. Performance model simulation support is needed for this goal, therefore input models for JSIM (i.e., the JMT model simulator [16]) shall be generated.

References

[1] Davide Arcelli and Vittorio Cortellessa. Software model refactoring based on performance analysis: better working on software or performance side? In *FESCA'13*, pages 33–47, February 2013.

[2] Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. Performance-driven software model refactoring. *Information & Software Technology*, 95:366–397, 2018.

[3] Jagdish Bansiya and Carl G Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[4] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, April 1975.

[5] Vittorio Cortellessa. Performance Antipatterns: State-of-Art and Future Perspectives. In *EPEW'13*, pages 1–6. Springer, 2013.

[6] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *SOSYM*, 13(1):391–432, 2014.

[7] Rahma Fourati, Nadia Bouassida, and Hanêne Ben Abdallah. A Metric-Based Approach for Anti-pattern Detection in UML Designs. *Computer and Information Science*, 364(Chapter 2):17–33, 2011.

[8] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using Graph Transformation for Practical Model-Driven Software Engineering. In *Model-Driven Software Development*, pages 91–117. Springer Berlin Heidelberg, 2005.

[9] Dimitris Kolovos, Louis Rose, Richard Paige, and Antonio Garcıa-Domınguez. *The epsilon book*. Structure, 2010.

[10] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. PerOpteryx: automated application of tactics in multi-objective software architecture optimization. In *QoSA'11*, pages 33–42. ACM, 2011.

[11] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance*. Prentice Hall, 1984.

[12] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *SoSyM*, 6(3):269–285, 2007.

[13] Mohammed Misbhauddin and Mohammad Alshayeb. UML model refactoring - a systematic literature review. *Empirical Software Engineering*, 20(1):206–251, 2015.

[14] Trevor Parsons and John Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–91, 2008.

[15] Trevor Parsons and John Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Tchnology*, pages 1–36, March 2008.

[16] G. Serazzi, G. Casale, M. Bertoli, G. Serazzi, G. Casale, and M. Bertoli. Java modelling tools: an open source suite for queueing network modelling and workload analysis. In *QEST'06*, pages 119–120, 2006.

[17] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns for identifying and correcting performance problems. In *CMGC'12*, 2012.

[18] Catia Trubiani, Anne Koziolek, Vittorio Cortellessa, and Ralf H. Reussner. Guilt-based handling of software performance antipatterns in palladio architectural models. *JSS*, 95:141–165, 2014.

[19] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In *QoSA'13*, pages 552–561. IEEE Press, 2013.

[20] Alexander Wert, Marius Oehler, Christoph Heger, and Roozbeh Farahbod. Automatic detection of performance anti-patterns in inter-component communications. In *QoSA'14*, pages 3–12. ACM, 2014.

[21] Jing Xu. Rule-based automatic software performance diagnosis and improvement. *Perf. Eval. Journal*, 69(11):525–550, November 2012.