Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 184 (2021) 541–548

Procedia Computer Science

www.elsevier.com/locate/procedia

The 12th International Conference on Ambient Systems, Networks and Technologies (ANT)
March 23 - 26, 2021, Warsaw, Poland

Envisioning Model-Based Performance Engineering Frameworks

Davide Arcelli[a]*

[a] *Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, via Vetoio 1, L'Aquila, 67100, Italy*

Abstract

Our daily activities depend on complex software systems that must guarantee certain performance. Several approaches have been devised in the last decade to validate software systems against performance requirements. However, software designers still encounter problems in the interpretation of performance analysis results (e.g., mean values, probability distribution functions) and in the definition of design alternatives (e.g., to split a software component in two and redeploy one of them) aimed at fulfilling performance requirements.

This paper describes a general model-based performance engineering framework to support designers in dealing with such problems aimed at enhancing the system. The framework relies on a formalization of the knowledge needed in order to characterize performance flaws and provide alternative system design. Such knowledge can be instantiated based on the techniques devised for interpreting performance analysis results and providing feedback to designers. Three techniques are considered in this paper for instantiating the framework and the main challenges to face during such process are pointed out and discussed.

Keywords: Software Performance Engineering; Model-Driven Engineering; Software Systems; Performance Analysis; Performance Optimization; Refactoring

1. Introduction

Over the last decade, research has highlighted the importance of performance as an essential quality attribute of many software systems, that is very complex and pervasive to be addressed. In fact, if performance targets are not met, a variety of negative consequences (such as user dissatisfaction, lost income, etc.) can impact on a significant fraction of software projects.

This motivates the activities of modeling and analyzing performance of software systems at the earlier phases of the life-cycle, by reasoning on predictive quantitative results, in order to avoid expensive post-deployment rework.

* Corresponding author.
E-mail address: davide.arcelli@gmail.com; davide.arcelli@univaq.it

Software Performance Engineering (SPE) is the discipline that represents the entire collection of engineering activities, used throughout the software development cycle, and directed to meet performance requirements [26]. In this context, many model-based performance engineering approaches have been proposed. Nevertheless, the identification of performance problems is still critical in the software design, mostly because the results of performance analysis (i.e. mean values, variances, and probability distributions) are difficult to be interpreted for providing feedback to designers (i.e. alternative software models) that can improve system performance. Support to the interpretation of performance analysis results that helps to fill the gap between numbers and design alternatives is still very challenging.

In this paper, a general model-based SPE framework is envisioned. To this aim, a knowledge is devised as basis for obtaining alternative software models, which enables a characterization of performance problems and corresponding solutions. After a general definition of the framework in formal terms, the latter is instantiated modulo three different well-known techniques, i.e. performance antipatterns [27], bottleneck analysis [29] and design space exploration [14]. Several challenges arise while instantiating the framework, which are also pointed out in this paper.

The paper is organized as follows. Section 2 illustrates and formalizes the model-based SPE framework. Section 3 shows how such framework can be instantiated relying on different techniques. Finally, Section 4 points out the main challenges in this context, whilst Section 5 concludes the paper.

2. General Framework

In this section we present a general framework that aims at suggesting alternative system design with enhanced performance, represented in Figure 1.

S represents a software model usually provided by the software designer. D represents a performance characterization of the system, such as workload and operational profile (i.e., the way users interact with the system), specified by a performance expert (possibly as software model annotations). R represents performance requirements. specified by the customer. Finally, P represents the performance indices of interests obtained through the performance analysis of S with respect to D.

If the calculated performance indices P satisfy the requirements R, that is $P \vDash R$, then the software system S and its performance characterization D fit the expectations. In this case no feedback is generated since no performance problems have been detected in the system under analysis.

On the contrary, if the performance requirements are not satisfied, then the framework interprets the analysis results P and generates software design feedback. Such step – namely *Results Interpretation and Feedback Generation* – produces a set of new Software Models S' owning new performance characterizations D'.

Identifying performance problems in the software design from performance analysis results is difficult since the latter are basically numbers (e.g., mean values, variances, etc.) or distribution functions, far from software design elements (e.g., software components, and deployment nodes) and design alternatives (asynchronous vs. synchronous communications). Moreover, performance indices often have to be jointly examined to localize the critical parts of a software model: a single performance index (e.g., the utilization of a service) might not give enough information without analyzing other indices (e.g., the throughput of a called service).

Specific experience and knowledge on software performance is needed to fill the gap between numbers and design alternatives, so to identify *problems* and devise *solutions* for them. Performance experts own such *knowledge*, namely K. The latter can be partitioned in two, i.e. $K = (K_p, K_s)$, where K_p represents the knowledge needed to identify the performance problems in the software model and K_s denotes the knowledge needed to devise design alternatives (possibly) implementing solutions to those problems. Conforming to this distinction, Results Interpretation and Feedback Generation step is composed by two tasks, as illustrated in the gray box at the bottom of Figure 1.

Results Interpretation is defined as a function $\textcircled{i}$ that takes as input: (i) a system design S, (ii) a performance characterization D of S and (iii) corresponding performance indices of interest P; by exploiting a problem characterization knowledge K_p, $\textcircled{i}$ identifies and returns a set of performance flaws {F} occurring to S. In lambda calculus notation [12], such function is formalized as:

$$\lambda K_p.\textcircled{i}(S, D, P) = \{F\} \tag{1}$$

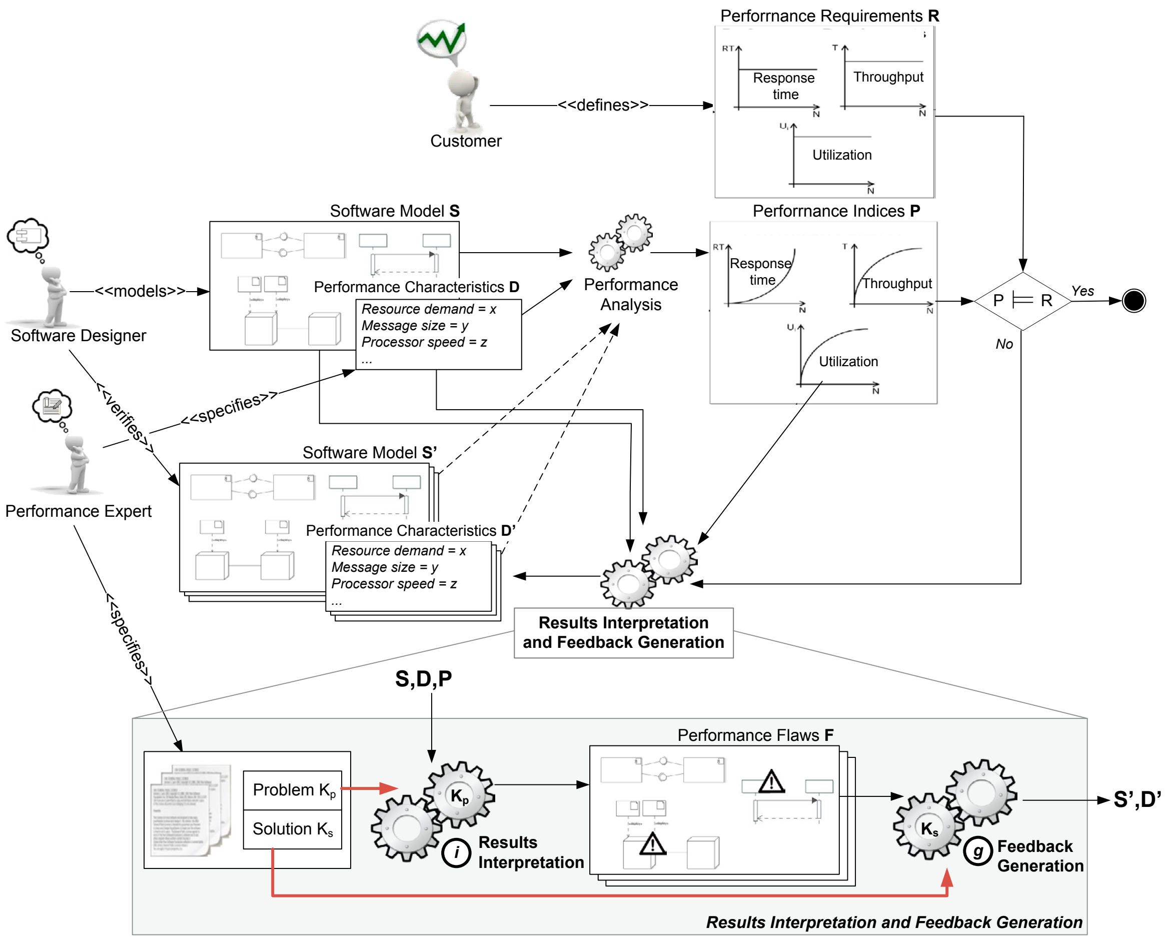

Fig. 1. Reference framework for results interpretation and feedback generation.

Feedback Generation is defined as a function ⓖ that takes as input: (i) a system design S, (ii) a performance characterization D of S and (iii) a set of performance flaws F occurring to S; by exploiting a solution characterization knowledge K_s, ⓖ suggests sets of refactoring actions which produce new software models S' characterized by new performance assumptions D' and showing new performance indices P'. In lambda calculus notation [12], such function is formalized as:

$$\lambda K_s.ⓖ(S, D, \{F\}) = \{(S', D', P')\} \tag{2}$$

The obtained software models S' can thus undergo results interpretation and new feedback can be generated. The final goal is to find a software model $S\star$ characterized by performance assumptions $D\star$ producing performance indices $P\star$ which best fit performance requirements. However, several iterations of the whole process can be conducted to find such a software model, because removing a flaw might introduce new flaws and does not necessarily result into performance improvement, due to heuristic decisions based on stochastic results.

3. Framework Instantiation and Examples of Existing Approaches

K_p and K_s strictly depend on the performance engineering technique(s) adopted for Result Interpretation and Feedback Generation. Table 1 describes K_p and K_s in the context of three techniques, namely Performance Antipatterns

[27], Bottleneck Analysis [29] and Design Space Exploration [14], and provides references to existing approaches exploiting such techniques.

Table 1. Examples of knowledge K for different performance engineering techniques and approaches exploiting them.

Technique	K_p	K_s	**Approaches**
Performance Antipatterns [27] (K^{PA})	Antipatterns specification describes the bad practices that negatively affects performance indices. One example of performance antipatterns is *Empty Semi Trucks* that occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.	Antipatterns specification includes a solution description that let the software architect devise refactoring actions. As an example *Empty Semi Trucks* can be solved by using the Batching performance pattern. Such pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.	[11, 8, 9]
Bottleneck Analysis [29] (K^{BA})	A system bottleneck occurs when one (or a limited number) of software or hardware resources are highly utilized and will be the first to saturate, thus throttling the system performance.	A system bottleneck is avoided by setting a multi-threading configuration for software components and a multi-processor configuration for hardware platforms, or re-allocating the work among the system resources.	[17, 30]
Design Space Exploration [14] (K^{DSE})	-	Explores the search space looking for solutions that best fit a predefined fitness function. Such function drives towards the sequence of refactoring actions to apply to this system, i.e., altering its design structure without altering its semantics with the final goal to locate reasonably good solutions.	[22, 3]

3.1. Performance Antipatterns

Performance antipatterns perfectly match a domain knowledge $K^{PA} = (K_p^{PA}, K_s^{PA})$ which distinguishes between problems and corresponding solutions, as they are two-fold: On one hand, a performance antipattern identifies a bad practice (namely K_p^{PA}) that negatively affects the performance indices of a software model; On the other hand, it provides descriptions of possible refactoring actions aimed at removing the bad practice (namely K_s^{PA}). Hence, bad practices define K_p^{PA}, whilst corresponding solutions define K_s^{PA}.

The function $\textcircled{i}$ in Equation 1 can be instantiated as $\textcircled{i^{PA}}$ in Equation 3, by exploiting Performance Antipatterns as backbone knowledge (i.e. $[K_p = K_p^{PA}]$). The resulting set of performance flaws, namely $\{F^{PA}\}$, is composed by instances of performance antipatterns detected on a software model S showing performance indices P under a certain characterization D.

$$\lambda K_p.\textcircled{i}(S, D, P)[K_p = K_p^{PA}] = \textcircled{i^{PA}}(S, D, P) = \{F^{PA}\} \tag{3}$$

Consequently, the function $\textcircled{g}$ in Equation 2 is re-written as $\textcircled{g^{PA}}$ in Equation 4 (i.e. $[K_s = K_s^{PA}, \{F\} = \{F^{PA}\}]$), resulting into alternative system design.

$$\lambda K_s.\textcircled{g}(S, D, \{F\})[K_s = K_s^{PA}, \{F\} = \{F^{PA}\}] = \textcircled{g^{PA}}(S, D, \{F^{PA}\}) = \{(S', D', P')\} \tag{4}$$

Many papers have appeared in literature which ground on performance antipatterns, mainly due to the fact that the latter are formulated in a general manner, hence they can be tailored to specific modelling notations. For example, in previous work [8] we have proposed an approach which allows to detect and remove performance antipatterns occurring to UML models [24] annotated with MARTE profile [25]. Such approach has brought to the development of a tool that has been exploited in [9], in the context of *MegaM@Rt*^{2} ECSEL project[1].

Moreover, as can be noticed from Table 1, design patterns may be used in model refactoring in order to remove performance antipatterns, that is what we have preliminary introduced in further previous work [11].

[1] https://megamart2-ecsel.eu/

3.2. Bottleneck Analysis

A bottleneck usually refers to hardware and/or software components as causes of performance degradation. It can be seen as a specialization of K^{PA}, where K_p and K_s characterize, respectively, possible bottleneck situations and corresponding solutions.

The function $\textcircled{i}$ in Equation 1 can be instantiated as $\textcircled{i^{BA}}$ in Equation 5, by exploiting Bottleneck Analysis as backbone knowledge (i.e. $[K_p = K_p^{BA}]$). The resulting set of performance flaws, namely $\{F^{BA}\}$, is composed by bottleneck components detected on a software model S showing performance indices P under a certain characterization D.

$$\lambda K_p.\textcircled{i}(S, D, P)[K_p = K_p^{BA}] = \textcircled{i^{BA}}(S, D, P) = \{F^{BA}\} \tag{5}$$

Consequently, the function $\textcircled{g}$ in Equation 2 is re-written as $\textcircled{g^{BA}}$ in Equation 6 (i.e. $[K_s = K_s^{BA}, \{F\} = \{F^{BA}\}]$), resulting into alternative system design.

$$\lambda K_s.\textcircled{g}(S, D, \{F\})[K_s = K_s^{BA}, \{F\} = \{F^{BA}\}] = \textcircled{g^{BA}}(S, D, \{F^{BA}\}) = \{(S', D', P')\} \tag{6}$$

Bottleneck analysis has been especially used until two decades ago in the context of software performance engineering. For example, Franks et al. [17] have developed a framework for detecting sources of bottlenecks which may occur to to Layered Queuing Network (LQN) models [18], applying improvements and estimating the effectiveness of the latter. Instead, Xu [30] has proposed an approach for the automated bottleneck identification and removal before the software system implementation, based on a set of rules for detecting bottlenecks occurring within LQNs.

3.3. Design Space Exploration

The function $\textcircled{i}$ in Equation 1 can be instantiated as $\textcircled{i^{DSE}}$ in Equation 7, by exploiting Design Space Exploration as backbone knowledge (i.e. $[K_p = K_p^{DSE}]$). However, as DSE does not rely on detecting particular situations in order to identify alternative system design, the knowledge K_p^{DSE} is empty, hence the resulting set of performance flaws – namely $\{F^{DSE}\}$ – is empty as well.

$$\lambda K_p.\textcircled{i}(S, D, P)[K_p = K_p^{DSE}] = \textcircled{i^{DSE}}(S, D, P) = \{F^{DSE}\} = \emptyset \tag{7}$$

Consequently, the function $\textcircled{g}$ in Equation 2 is re-written as $\textcircled{g^{DSE}}$ in Equation 8 (i.e. $[K_s = K_s^{DSE}, \{F\} = \{F^{DSE}\} = \emptyset]$), resulting into alternative system design.

$$\lambda K_s.\textcircled{g}(S, D, \{F\})[K_s = K_s^{DSE}, \{F\} = \{F^{DSE}\} = \emptyset] = \textcircled{g^{DSE}}(S, D, \emptyset) = \{(S', D', P')\} \tag{8}$$

Design Space Exploration has started to be applied to performance optimization of hardware/software systems since a decade, i.e. when Martens et al. [22] have used NSGA-II genetic algorithm [16] – that is a widespread evolutionary computation technique – in order to find optimal trade-offs among performance, reliability and costs, for software architectures modelled in terms of Palladio Component Models [13].

NSGA-II has been recently used in previous work [3], where an approach has been proposed for optimizing the performance of self-adaptive systems represented by a particular type of QN models [20], namely SMAPEA QNs [5, 4].

3.4. Combined Approaches

Performance engineering techniques can be also used in synergy, as demonstrated by some approaches that have appeared in literature. For example, Trubiani et al. [28] have proposed an approach which exploits K^{PA} and K^{BA} in different ways, i.e.:

1. By exploiting K_s^{PA} and K_s^{BA} separately, meaning that *Results Interpretation and Feedback Generation* step is executed twice, i.e. one per each knowledge, and the obtained performance improvements are compared in order to provide evidence of the relative strengths and weaknesses of the two techniques.
2. By exploiting one knowledge first – e.g. K_p^{BA} and K_s^{BA} – followed by the other one, i.e. *Results Interpretation and Feedback Generation* step is firstly executed with respect to the former knowledge and the obtained output might be subsequently improved by further execution with respect to the latter knowledge.
3. By exploiting the intersection between K_p^{BA} and K_p^{PA} in order to drive the application of K_s^{PA} during *Results Interpretation and Feedback Generation*, meaning that only performance antipatterns involving bottlenecks are solved thus reducing the solution space.

More recently, we have introduced EASIER [7], that is an approach relying on NSGA-II genetic algorithm – i.e. K^{DSE} – for refactoring software architecture modelled in terms of Æmilia specifications [1], aimed at multi-objective performance optimization. The number of performance antipatterns occurring to the system is one of the considered performance metrics composing the multi-objective fitness function, thus combining K_p^{PA} and K^{DSE}; however, refactoring is not aimed at removing antipatterns – i.e. K_p^{PA} is not exploited – although this might happen as a side-effect.

4. Challenges

Several challenging aspects have to be considered while instantiating the performance-based model refactoring framework, which are pointed out in the following.

Ambiguity in formalization. Different knowledge formalizations bring to different results. Hence, an unavoidable gap arises among different interpretations of the existing literature used to build up a knowledge. The consolidation of the devised knowledge requires wide investigation while applying the framework. Moreover, the formalization also depends on the system modeling language. For example, dealing with annotated UML software models might allow to formalize fine-grained knowledge, whilst Queuing Networks might restrict the set of available constructs, resulting into a coarse-grained knowledge [6].

Influence of other software layers. Often a software model only contains information on the software system and the hardware platform. However, between these two layers there might other components, e.g. middleware, which should be considered because they could jeopardize the performance of the whole system [23].

Accuracy of problem identification. False positive/negative problem instances may be detected, mostly due to the presence of thresholds in the function ⓘ, which inevitably introduce a degree of uncertainty influencing recall and precision [10]. Potential sources to suitably tune thresholds are: *(i)* the system requirements; *(ii)* the domain expert's knowledge; *(iii)* the evaluation of the system under analysis.

Inhibition of refactoring actions and conflicts. Restrictions due to functional or other non-functional requirements may inhibit refactoring actions, e.g. splitting or redeploying specific components due to legacy constraints or budget limitations. Moreover, model refactoring cannot be unambiguously applied in case of incoherence of the corresponding refactoring actions, e.g. changing a property of a system component which has been previously removed. Hence, particular mechanisms should be introduced – e.g. pre- and post- conditions [21] – enabling the specification of criteria which allow to properly combine refactoring actions.

Model transformation. Working on software models may require model transformation capabilities. In fact, a system may be represented by a set of models, each characterizing a different "views" (e.g. behavioral, deployment); in this context, consistency among the different views of the system shall be maintained by means of advanced Model-Driven Engineering techniques [19], as the refactoring of a view shall properly propagate to other views. Moreover, model transformation may be required in order to obtain a performance model from the software model, e.g. from UML to LQN [2], aimed at estimating performance indices of interest and possibly enabling performance flaws detection and removal.

No strict guarantee of performance improvement. Producing a candidate software model S' by solving one or more performance flaws or by automatic generation does not imply performance improvement, as only further performance analysis (i.e. P') can reveal if S' shows enhanced performance. Moreover, when a candidate software model S' is identified, it might satisfy some performance requirements while worsening some others. Hence, trade-offs among performance requirements shall be taken into account, as well as among other non-functional properties, e.g. reliability, dependability, costs [22].

Lack of model parameters. Performance engineering is not limited along the software life-cycle [15], but it is obvious that an early usage is subject to lack of information because the system knowledge improves while the development process evolves. The framework should be thus applied in different phases of the software life-cycle, as the knowledge improves and prototype system implementations are available.

Run-time metrics. A performance-driven evolution of a software system at run-time may be required. To this aim, proper run-time traces may be generated from a running system implementation in order to obtain realistic performance indices [9]. However, processing time restrictions at run-time might bring to re-think/adapt/tailor the techniques used to recommend alternative system designs/configurations, preferring solutions which are not optimal, but that keep the system within a feasible behavioral envelope.

Level of automation and human need. A key point for adopting the framework in practice is the degree of automation that can be achieved. Automation should not fully replace the human contribution; in fact, human intervention may be needed for building the knowledge – e.g. defining performance antipatterns detection rules and corresponding solutions – and to drive the process of result interpretation and feedback generation. This makes the framework suitable to be adopted as a semi-automated support for identifying good candidate software models.

5. Conclusion

In this paper, a general model-based performance engineering framework aimed at enhancing system design has been envisioned, focusing on the knowledge used as basis for producing alternative software models, which enables a characterization of performance problems and corresponding solutions, depending on different techniques. Such knowledge has been here formalized in general terms and then instantiated with respect to three different well-known techniques, i.e. performance antipatterns, bottleneck analysis and design space exploration.

Examples of existing approaches exploiting each of these techniques or a mix of them have been provided as well, and the main challenges to face while instantiating the framework have been pointed out, highlighting the complexity of model-based software performance engineering.

Acknowledgements

This work is supported by the Italian Ministry of Education, University and Research – MIUR, L. 297, art. 10.

References

[1] Aldini, A., Corradini, F., Bernardo, M., 2010. Component-Oriented Performance Evaluation. Springer London, London. chapter A Process Algebraic Approach to Software Architecture Design. pp. 203–238. doi:10.1007/978-1-84800-223-4_6.

[2] Altamimi, T., Zargari, M.H., Petriu, D.C., 2016. Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links, in: Mindel, M., Jones, B., Müller, H.A., Onut, V. (Eds.), Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON 2016, Toronto, Ontario, Canada, October 31 - November 2, 2016, IBM / ACM. pp. 208–217. URL: http://dl.acm.org/citation.cfm?id=3049899.

[3] Arcelli, D., 2020a. A multi-objective performance optimization approach for self-adaptive architectures, in: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (Eds.), Software Architecture - 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14-18, 2020, Proceedings, Springer. pp. 139–147. doi:10.1007/978-3-030-58923-3_9.

[4] Arcelli, D., 2020b. A novel family of queuing network models for self-adaptive systems, in: Hammoudi, S., Pires, L.F., Selic, B. (Eds.), Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, Revised Selected Papers, Springer. pp. 349–376. doi:10.1007/978-3-030-67445-8_15.

[5] Arcelli, D., 2020c. Towards a generalized queuing network model for self-adaptive software systems, in: Hammoudi, S., Pires, L.F., Selic, B. (Eds.), Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, SCITEPRESS. pp. 457–464. doi:10.5220/0009180304570464.

[6] Arcelli, D., Cortellessa, V., 2013. Software model refactoring based on performance analysis: better working on software or performance side?, in: Buhnova, B., Happe, L., Kofron, J. (Eds.), Proceedings 10th International Workshop on Formal Engineering Approaches to Software Components and Architectures, FESCA 2013, Rome, Italy, March 23, 2013, pp. 33–47. doi:10.4204/EPTCS.108.3.

[7] Arcelli, D., Cortellessa, V., D'Emidio, M., Di Pompeo, D., 2018a. EASIER: an evolutionary approach for multi-objective software architecture refactoring, in: IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018, IEEE Computer Society. pp. 105–114. doi:10.1109/ICSA.2018.00020.

[8] Arcelli, D., Cortellessa, V., Di Pompeo, D., 2018b. Performance-driven software model refactoring. Inf. Softw. Technol. 95, 366–397. doi:10.1016/j.infsof.2017.09.006.

[9] Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., Tucci, M., 2019. Exploiting architecture/runtime model-driven traceability for performance improvement, in: IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019, IEEE. pp. 81–90. doi:10.1109/ICSA.2019.00017.

[10] Arcelli, D., Cortellessa, V., Trubiani, C., 2013. Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns. Electronic Communications of the EASST 59. doi:10.14279/tuj.eceasst.59.937.

[11] Arcelli, D., Di Pompeo, D., 2017. Applying design patterns to remove software performance antipatterns: A preliminary approach, in: Shakshuki, E.M. (Ed.), The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017) / The 7th International Conference on Sustainable Energy Information Technology (SEIT 2017), 16-19 May 2017, Madeira, Portugal, Elsevier. pp. 521–528. doi:10.1016/j.procs.2017.05.330.

[12] Barendregt, H.P., 1991. The Lambda Calculus. Number 103 in Studies in Logic and the Foundations of Mathematics. revised ed., North-Holland, Amsterdam.

[13] Becker, S., Koziolek, H., Reussner, R.H., 2009. The palladio component model for model-driven performance prediction. J. Syst. Softw. 82, 3–22. doi:10.1016/j.jss.2008.03.066.

[14] Cardoso, J.M., Coutinho, J.G.F., Diniz, P.C., 2017. Chapter 8 - additional topics, in: Cardoso, J.M., Coutinho, J.G.F., Diniz, P.C. (Eds.), Embedded Computing for High Performance. Morgan Kaufmann, Boston, pp. 255 – 280. doi:10.1016/B978-0-12-804189-5.00008-9.

[15] Cortellessa, V., Di Marco, A., Inverardi, P., 2011. Software lifecycle and performance analysis, in: Model-Based Software Performance Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 65–77. doi:10.1007/978-3-642-13621-4_4.

[16] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation 6, 182–197. doi:10.1109/4235.996017.

[17] Franks, G., Petriu, D.C., Woodside, C.M., Xu, J., Tregunno, P., 2006. Layered bottlenecks and their mitigation, in: Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA, IEEE Computer Society. pp. 103–114. doi:10.1109/QEST.2006.23.

[18] Franks, G., Woodside, C.M., 2004. Multiclass multiservers with deferred operations in layered queueing networks, with software system applications, in: DeGroot, D., Harrison, P.G., Wijshoff, H.A.G., Segall, Z. (Eds.), 12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 4-8 October 2004, Vollendam, The Netherlands, IEEE Computer Society. pp. 239–248. doi:10.1109/MASCOT.2004.1348262.

[19] Kretschmer, R., Khelladi, D.E., Lopez-Herrejon, R.E., Egyed, A., 2020. Consistent change propagation within models. J. Softw. Syst. Modeling doi:10.1007/s10270-020-00823-4.

[20] Lazowska, E.D., Zahorjan, J., Sevcik, K.C., 1986. Computer system performance evaluation using queueing network models. Annual Review of Computer Science 1, 107–137. doi:10.1146/annurev.cs.01.060186.000543.

[21] Mansoor, U., Kessentini, M., Wimmer, M., Deb, K., 2017. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Softw. Qual. J. 25, 473–501. doi:10.1007/s11219-015-9284-4.

[22] Martens, A., Koziolek, H., Becker, S., Reussner, R.H., 2010. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms, in: Adamson, A., Bondi, A.B., Juiz, C., Squillante, M.S. (Eds.), Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010, ACM. pp. 105–116. doi:10.1145/1712605.1712624.

[23] Rafique, A., Van Landuyt, D., Lagaisse, B., Joosen, W., 2018. On the performance impact of data access middleware for nosql data stores a study of the trade-off between performance and migration cost. IEEE Transactions on Cloud Computing 6, 843–856. doi:10.1109/TCC.2015.2511756.

[24] Selic, B., 2006. Uml 2: a model-driven development tool. IBM Syst. J. 45, 607–620. doi:10.1147/sj.453.0607.

[25] Selic, B., Grard, S., 2013. Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] Smith, C.U., 2007. Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures. Springer Berlin Heidelberg, Berlin, Heidelberg. chapter Introduction to Software Performance Engineering: Origins and Outstanding Problems. pp. 395–428. doi:10.1007/978-3-540-72522-0_10.

[27] Smith, C.U., Williams, L.G., 2003. More new software antipatterns: Even more ways to shoot yourself in the foot, in: 29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings, Computer Measurement Group. pp. 717–725.

[28] Trubiani, C., Di Marco, A., Cortellessa, V., Mani, N., Petriu, D., 2014. Exploring synergies between bottleneck analysis and performance antipatterns, in: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, New York, NY, USA. p. 75–86. doi:10.1145/2568088.2568092.

[29] Wescott, B., 2013. Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With. 1st ed., CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[30] Xu, J., 2010. Rule-based automatic software performance diagnosis and improvement. Performance Evaluation 67, 585 – 611. doi:10.1016/j.peva.2009.07.004.