VisArch: Visualisation of Performance-based Architectural Refactorings*

Catia Trubiani[1][0000−0002−7675−6942], Aldeida Aleti[2][0000−0002−1716−690X], Sarah Goodwin[2][0000−0001−8894−8282], Pooyan Jamshidi[3][0000−0002−9342−0703], Andre van Hoorn[4][0000−0003−2567−6077], and Samuel Gratzl[5][0000−0002−3712−8660]

[1] Gran Sasso Science Institute, Italy, catia.trubiani@gssi.it
[2] Monash University, Australia, {aldeida.aleti, sarah.goodwin}@monash.edu
[3] University of South Carolina, US, PJAMSHID@cse.sc.edu
[4] University of Stuttgart, Germany, van.hoorn@informatik.uni-stuttgart.de
[5] https://www.sgratzl.com, samuel_gratzl@gmx.at

Abstract. Evaluating the performance characteristics of software architectures is not trivial since many factors, such as workload fluctuations and service failures, contribute to large variations. To reduce the impact of these factors, architectures are refactored so that their design becomes more robust and less prone to performance violations. This paper proposes an approach for visualizing the impact, from a performance perspective, of different performance-based architectural refactorings that are inherited by the specification of performance antipatterns. A case study including 64 performance-based architectural refactorings is adopted to illustrate how the visual representation supports software architects in the evaluation of different architecture design alternatives.

Keywords: Software Architecture · Performance · Visualisation

1 Introduction

Performance evaluation of software architectures is a complex activity, even more so when workload fluctuations and software/hardware failures contribute to distributions in requests and resources' availability [9, 14]. These variabilities may be smoothed by equipping the architecture with a portfolio of refactoring actions to make it more robust [2,6], i.e., less prone to performance issues. Performance-based architectural refactorings are behaviour-preserving actions [10] that may span in multiple dimensions, such as design changes and/or redeployment, hardware settings, communication patterns among software components, etc. [16,17].

Understanding what are the most suitable performance-based architectural refactorings is indeed not trivial since there might be several trade-off decisions arising in the evaluation, and software architects are usually not supported in this task. To get system performance improvements, we make use of *software performance antipatterns* [22] since they have been applied in the context of software architectures and shown to be beneficial [24]. The main benefit of adopting

* This work has been partially supported by the MIUR PRIN project SEDUCE 2017TWRCNB and the Baden-Württemberg Stiftung.

antipatterns is that their specification includes reusable solutions that can be applied across different domains, e.g., very recently performance antipatterns have been investigated for Cyber-Physical Systems [21]. Moreover, to deal with system uncertainties, *polynomial chaos expansion* has been applied [2] for computing the cumulative distributions related to performance metrics of interest that are known to be affected by uncertain parameters.

In the literature, the problem of optimizing the non-functional characteristics of software architectures, even under uncertainty, has been tackled by several approaches [1,4,5,7,9,18]. However, most of the developed methodologies focus on a specific modeling and/or analysis formalism (e.g., fuzzy logic [9]). As opposite, to the best of our knowledge, there is limited work in the field of visualising non-functional (e.g., performance) data and its match with system architectural choices. The interest of the research community in software and performance visualisation is growing [3,19], and the state-of-the-art for performance visualisation techniques has been preliminary evaluated in [15].

This paper investigates the effectiveness of interactive visualisation techniques [12,13] in the selection of design alternatives. Our research question is:

How can visualisation help identify refactorings that improve performance? To answer this question, we design a case study of 64 architectural refactorings by extending previous work [24]) and investigate how visualisation supports the evaluation of the impact of these refactorings on performance metrics, such as system response time, service throughput, and resource utilization.

The main contribution of this paper is the VisArch visualisation approach, applied to a case study, that supports: (i) the evaluation of performance-based architectural refactorings (with a focus on performance antipatterns); (ii) the estimation of the uncertainty propagation by measuring the impact of an architectural change on system robustness (based on polynomial chaos expansion); (iii) the exploitation of the uncertainty and robustness estimates by software architects for their decision making in the selection of design alternatives.

The rest of the paper is organized as follows. Section 2 describes the details of our approach. Section 3 briefly discusses the case study and illustrates the visualisation results. Section 4 concludes the paper by outlining future research directions. All artifacts are publicly available [23].

2 VisArch: Visualising architectural refactorings

In this paper, we propose an approach to visualise the impact, from a performance-based perspective, of different architectural refactorings. The challenge is to keep track of the interweaving ways in which a refactoring action at the architectural level may impact the system performance. Our approach, called *VisArch*, aims to address this complexity, by leveraging the benefits of data visualisation to help with assessing the impact of refactoring techniques. Figure 1 provides an overview of the workflow we follow to apply the VisArch approach. Input/output artifacts and operational steps are described in the following.

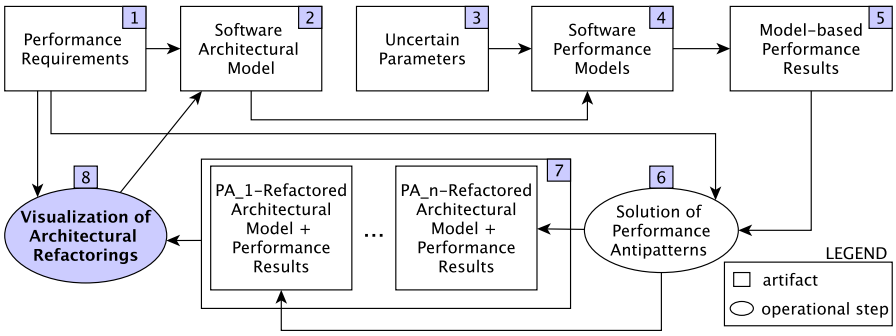


Fig. 1. Overview of VisArch.

Performance Requirements – see the box labeled as 1 in Figure 1 – represent the required performance characteristics, e.g., the system response time has to be less than 10 seconds. *Software Architectural Model* (2) represents the system in terms of software components, their interactions, and deployment settings. *Uncertain Parameters* (3) represent the system characteristics that are unknown, parameter values are expressed as distribution functions (e.g., uniform, normal, discrete, triangular). For example, the workload can be specified as $[workload : Distribution = (UNIFORM, 100, 150)]$, meaning that number of users varies with a uniform distribution between 100 and 150. This allows a flexible specification of uncertain parameters and also captures their diverse nature. *Software Performance Models* (4) represent the abstractions of the system to derive its performance characteristics. Several performance models have been developed in the literature, and we use Layered Queuing Networks (LQNs) [8], since such models have been demonstrated to suitably approximate real-world scenarios [11]. *Model-based Performance Results* (5) represent the predictions of the performance characteristics of an application, such as system response time (RT), throughput (TH) of services, and hardware utilization (U), via analytical models. In our case, such results are obtained by adopting well-known solution techniques within the LQN solver [11]. *Solution of Performance Antipatterns* (6) takes as input performance requirements and model-based performance results that are compared. In case of requirements' violations, the generation of architectural refactorings is supported by the solution of performance antipatterns, since they have been demonstrated to be beneficial in the context of software architectures [24]. *PA_x-Refactored Architectural Model + Performance Results* (7) represent the set of architectural models that are generated after solving the performance antipatterns. Even if not reported in Figure 1, all the generated architectural models are transformed into LQN models and analyzed. Performance results are then coupled with the corresponding architectural models, since they will be used for visualisation purposes.

visualisation of Architectural Refactorings (8) represents the main contribution of this paper. Performance results are presented using a new interactive

visualisation technique for exploring heterogeneous multi-attribute rankings, i.e., LineUp [12][6]. This technique allows the large amount of performance data including the uncertainties to be presented as an intuitive visual overview. The technique can be described as an interactive table, consisting of visualisations in rows and columns that can be quickly filtered and reordered. It uses the concept of small-multiple visualisations [25] to provide first a overview, then analytical details on demand [20]. Performance results are loaded into LineUp, and a visualisation is built. Each row represents one possible refactoring combination that has been tested. Individual columns contain the results of the different samples (resulting from the specification of uncertain parameters), which are presented as boxplots to highlight the underlying distribution of each sample. Rows and columns can be sorted on the basis of whether a performance antipattern has been refactored or not (see Section 3). This presents a visual overview showing the refactoring impact on certain performance requirements. The visualisation provides an instrument to clearly recognize the most suitable architectural alternatives when analyzing the performance requirements of interest.

3 Visualisation Results

The visualisation approach is illustrated by means of a case study presented in [24], namely the Book and Movie online-Shop (B&M-S). Figure 2 depicts an excerpt of the software architectural model where the main software components and hardware machines are shown (legend at the top right). Table 1 briefly reports the performance-based architectural refactorings (AR), inherited from [24] and used for the visualisation. The complete description of the case study is reported in our supplementary material [23].

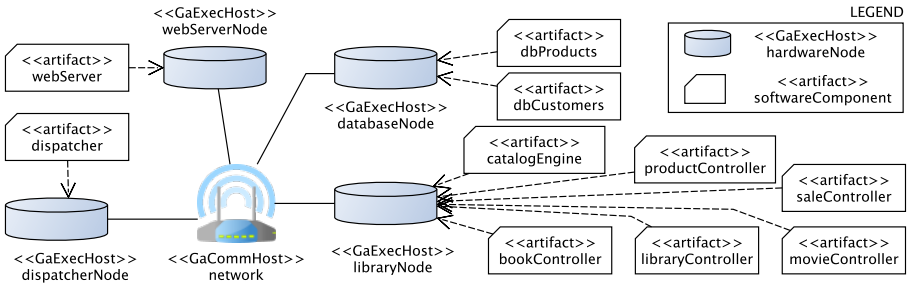


Fig. 2. B&M-S: excerpt of the Software Architectural Model.

We applied the LineUp [12] technique to visualise the architectural refactorings. Figures 3–4 depict the detailed outcomes. The first column indicates which refactoring action has been applied to the initial software architecture. The refactored solutions are associated to the following colours:

AR_1-CTH	**Red**	AR_2-CPS_{db}	**Purple**	AR_3-$BLOB$	**Brown**	AR_4-EP	**Pink**
AR_5-EST	**Olive**	AR_6-CPS_{lib}	**Teal**				

[6] `https://lineup.js.org`

Table 1. Performance-based architectural refactorings (AR) driven by the solution of performance antipatterns.

AR_1	Solving the CTH (Circuitous Treasure Hunt) performance antipattern. To better balance the load between the *saleController* and *dbCustomers* components, the latter is invoked twice. Its demand increases (i) to check user credentials from 0.03 to 0.09, (ii) to verify customer promotions from 0.03 to 0.06.
AR_2	Solving the CPS_{db} (Concurrent Processing System detected on the database component) performance antipattern. To better balance the resources, the *dbCustomers* component is redeployed from *databaseNode* to the *dispatcherNode*.
AR_3	Solving the $BLOB$ (God class/component) performance antipattern. To better balance the load between *libraryController* vs. *bookLibrary* and *movieLibrary* components, these latter components have been redesigned. The resource demand of *libraryController* decreases from 0.05 to 0.02, whereas the demands of *bookLibrary* and *movieLibrary* both increase from 0.03 to 0.045.
AR_4	Solving the EP (Extensive Processing) performance antipattern. A newly added component, namely *catalogEngineMirror* handles generic and expensive catalogs, whereas book and movies catalogs are handled by the *catalogEngine* component.
AR_5	Solving the EST (Empty Semi Trucks) performance antipattern. To better balance the load between the *saleController* and *productController* components, the computation is moved to this latter component that is invoked once to check the quality of products, and consequently its demand increases from 0.01 to 0.03.
AR_6	Solving the CPS_{lib} (Concurrent Processing System detected on the library component) performance antipattern. To optimise the resources, the *libraryController* component is redeployed from *libraryNode* to the *dispatcherNode*.

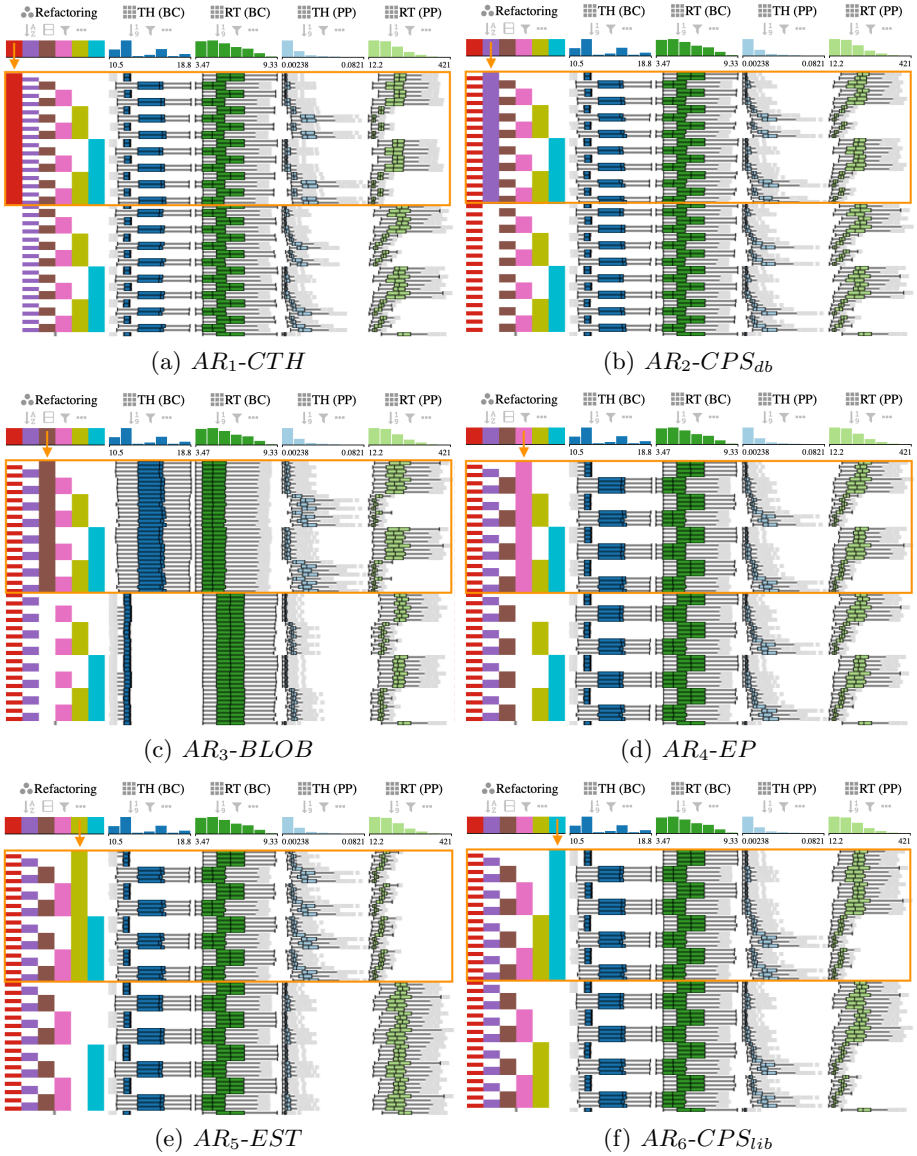
The impact of the refactoring solution can be inspected visually via the performance metrics we consider, specifically the throughput (TH) and response time (RT) of these services: *Browse Catalogue* (BC), and *Purchase Product* (PP).

TH(BC) (**Blue**), RT(BC) (**Green**), TH(PP) (**Light Blue**) and RT(PP) (**Light Green**)

The distribution of the results for each of the four metrics are shown in separate columns as a histogram at the top, indicating the total distribution of the results, and as boxplots in each row, depicting the distribution of the metrics for each combination of refactoring technique.

Figure 3a, the top half of the table (highlighted orange), depicts solutions where the CTH was applied, whereas the bottom half shows solutions where this refactoring was not applied. As we can see, CTH does not have a significant impact on any of the performance metrics, since the visual patterns of the performance metric columns are similar. The same holds true for the refactoring techniques: CPS_{db} in Figure 3b, EP in Figure 3d, and CPS_{lib} in Figure 3f. Yet, the visualisations reveal a significant impact on the performance metrics, both in terms of variance and mean, for the refactoring techniques that address $BLOB$ and EST performance antipatterns, shown in Figures 3c and 3e.

The biggest impact of refactoring that addresses $BLOB$ is on throughput, and is more evident on TH(BC). Figure 3c shows that the solutions depicted in the top half of the graph have on average a higher throughput, i.e., refactoring



(a) AR_1-CTH (b) AR_2-CPS_{db}

(c) AR_3-$BLOB$ (d) AR_4-EP

(e) AR_5-EST (f) AR_6-CPS_{lib}

Fig. 3. Visualisations of performance-based architectural refactorings. Orange highlights reordering and the solutions where performance-based refactoring was applied. All figures include 64 rows representing the tested refactoring combinations.

BLOB improves mean throughput. However, the variance of the throughput is quite high for all solutions, as shown by the wider boxplots compared to solutions without this refactoring. This means that the throughput in the refactored solutions is likely to fluctuate more, making the system unstable.

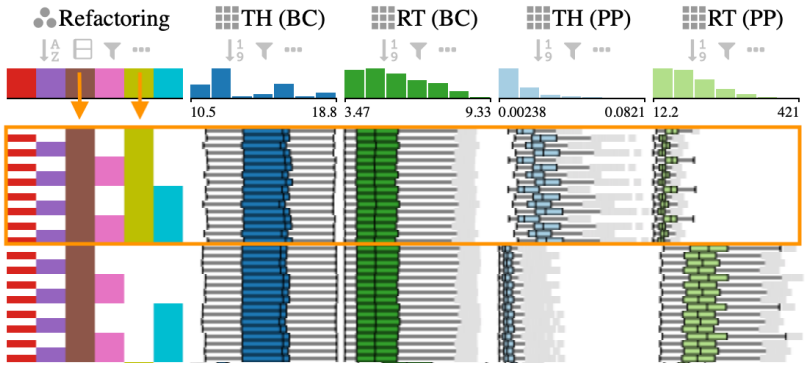


Fig. 4. Visualisation of $AR_{3.5}$- *BLOB* and *EST* architectural refactorings. For sake of space, the first 32 rows are showed and represent a subset of tested refactorings.

A similar scenario is shown for TH(PP) in Figure 3c, however the impact of refactoring BLOB is not as strong, and there seems to be an *interaction* between *BLOB* (brown) and *EST* (olive). The effect of *EST* on TH(PP) is more evident in Figure 3e, where rows have been arranged to show solutions fixing this antipattern at the top of the graph. Refactoring for *EST* improves TH(PP), and has a slight negative impact on its variance, i.e., refactored solutions are not as stable in terms of TH(PP). However, the impact on variance by refactoring *EST* is not as bad as the impact of BLOB on TH(BC). *EST* does not impact TH(BC) in any way. Interestingly, refactoring *EST* improves RT as well, both in terms of reducing mean RT(PP) and its variance. *EST* is the only refactoring technique that improves RT(PP). *BLOB* had a slight positive effect on RT(BC), but this was not as significant as the impact on RT(PP) by applying *EST*.

To further investigate the *interaction* between *BLOB* and *EST*, we order the results based on both columns, see Figure 4. The visualisation clearly shows that all performance metrics benefit from it, although in some metrics, such as RT(BC), the improvement is not as significant. Summarising, *BLOB* improves both throughput metrics, i.e., TH(BC) and TH(PP), but the stability of the system became worse. *BLOB* also has a negligible impact on response time, by slightly improving RT(BC) and no impact on RT(PP). *EST* improves the distribution of TH(PP) and RT(PP), but has no impact on TH(BC) and RT(BC).

4 Conclusion

In this paper we presented an approach to visualise performance-based architectural refactorings. Our experimentation mainly focused on the feasibility of applying visualisation techniques in the context of evaluating the performance of different architectural alternatives. As future work, we plan to quantify the gain for software architects through a user study to learn how do they perceive visualisation as support for the actual selection of design alternatives. Moreover, we plan to explore further visualisation techniques and apply the approach to more complex case studies, possibly from an industrial context.

References

1. Aleti, A., et al.: Software architecture optimization methods: A systematic literature review. IEEE Trans. Software Eng. **39**(5), 658–683 (2013)
2. Aleti, A., et al.: An efficient method for uncertainty propagation in robust software performance estimation. Journal of Systems and Software **138**, 222–235 (2018)
3. Beck, F., et al.: Visualizing systems and software performance - Report on the GI-Dagstuhl seminar (2018), `https://peerj.com/preprints/27253/`
4. Berrevoets, R., Weyns, D.: A qos-aware adaptive mobility handling approach for lora-based iot systems. In: SASO. pp. 130–139 (2018)
5. Busch, A., et al.: Assessing the Quality Impact of Features in Component-Based Software Architectures. In: ECSA. pp. 211–219 (2019)
6. Calinescu, R., et al.: Designing robust software systems through parametric markov chain synthesis. In: ICSA. pp. 131–140 (2017)
7. Cámara, J., et al.: Synthesis and quantitative verification of tradeoff spaces for families of software systems. In: ECSA. pp. 3–21 (2017)
8. Das, O., Woodside, C.M.: Analyzing the effectiveness of fault-management architectures in layered distributed systems. Perform. Eval. **56**(1-4), 93–120 (2004)
9. Esfahani, N., et al.: Guidearch: guiding the exploration of architectural solution space under uncertainty. In: ICSE. pp. 43–52 (2013)
10. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
11. Franks, G., et al.: Enhanced modeling and solution of layered queueing networks. IEEE Trans. Software Eng. **35**(2), 148–161 (2009)
12. Furmanova, K., et al.: Taggle: Combining overview and details in tabular data visualizations. Information Visualization **19**(2), 114–136 (2020)
13. Goodwin, S., et al.: What do Constraint Programming Users Want to See? Exploring the Role of Visualisation in Profiling of Models and Search. IEEE Trans. Vis. Comput. Graph. **23**(1), 281–290 (2017)
14. Incerto, E., et al.: Software performance self-adaptation through efficient model predictive control. In: ASE. pp. 485–496 (2017)
15. Isaacs, K.E., et al.: State of the Art of Performance Visualization. In: EuroVis - STARs. The Eurographics Association (2014)
16. Jamshidi, P., et al.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: ASE. pp. 497–508 (2017)
17. Jamshidi, P., Casale, G.: An uncertainty-aware approach to optimal configuration of stream processing systems. In: MASCOTS. pp. 39–48 (2016)
18. Mahdavi-Hezavehi, S., et al.: A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. Inf. Softw. Technol. **90**, 1–26 (2017)
19. Okanovic, D., et al.: Concern-driven reporting of software performance analysis results. In: ICPE. pp. 1–4. ACM (2019)
20. Shneiderman, B.: The eyes have it: a task by data type taxonomy for information visualizations. In: Symposium on Visual Languages. pp. 336–343 (1996)
21. Smith, C.U.: Software performance antipatterns in cyber-physical systems. In: ICPE. pp. 173–180 (2020)
22. Smith, C.U., Williams, L.G.: Software performance antipatterns for identifying and correcting performance problems. In: CMG. pp. 717–725 (2012)
23. Trubiani, C., et al.: Artifacts, `https://doi.org/10.5281/zenodo.3936656`
24. Trubiani, C., et al.: Exploring synergies between bottleneck analysis and performance antipatterns. In: ICPE. pp. 75–86 (2014)
25. Tufte, E.: Envisioning Information. Graphics Press, Cheshire, CT, USA (1990)