Exploring Synergies between Bottleneck Analysis and Performance Antipatterns

Catia Trubiani, Antinisca Di Marco, Vittorio Cortellessa
University of L'Aquila, Italy
{catia.trubiani, antinisca.dimarco, vittorio.cortellessa}@univaq.it

Nariman Mani, Dorina Petriu
Carleton University, Ottawa, Canada
{nmani, petriu}@sce.carleton.ca

ABSTRACT

The problem of interpreting the results of performance analysis is quite critical, mostly because the analysis results (i.e. mean values, variances, and probability distributions) are hard to transform into feedback for software engineers that allows to remove performance problems. Approaches aimed at identifying and removing the causes of poor performance in software systems commonly fall in two categories: (i) bottleneck analysis, aimed at identifying overloaded software components and/or hardware resources that affect the whole system performance, and (ii) performance antipatterns, aimed at detecting and removing common design mistakes that notably induce performance degradation. In this paper, we look for possible synergies between these two categories of approaches in order to empower the performance investigation capabilities. In particular, we aim at showing that the approach combination allows to provide software engineers with broader sets of alternative solutions leading to better performance results. We have explored this research direction in the context of Layered Queueing Network models, and we have considered a case study in the e-commerce domain. After comparing the results achievable with each approach separately, we quantitatively show the benefits of merging bottleneck analysis and performance antipatterns.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques, Performance Attributes; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Design.

Keywords

Software Performance; Model-based Performance Analysis; Bottleneck Analysis; Performance Antipatterns; Software Performance Feedback.

ICPE'14, March 22–26, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2568088.2568092.

1. INTRODUCTION

In the software development domain there is a very high interest in the early validation of performance requirements because this ability avoids late and expensive fixes to consolidated software artifacts.

Model-based approaches, pioneered under the name of Software Performance Engineering (SPE) by Smith [24, 26, 23], aim at producing performance models early in the development cycle and using quantitative results from model solutions to refactor the architecture and design [17] with the purpose of meeting performance requirements [27].

Advanced Model-Driven Engineering (MDE) techniques have successfully been used in the last few years to introduce automation in software performance modeling and analysis [5, 13, 7]. Nevertheless, the problem of interpreting the performance analysis results is still quite critical. A large gap exists between the representation of performance analysis results and the feedback expected by software architects. For instance, the results contain numbers (e.g., mean response time, throughput, utilization, variance, etc.), whereas the feedback should include architectural suggestions, i.e., design alternatives, useful to overcome performance problems (e.g., split a software component in two components and re-deploy one of them).

Figure 1 shows the process we propose for merging bottleneck analysis and performance antipatterns in a round-trip SPE process. Ovals in the figure represent operational steps whereas square boxes represent input/output data. Vertical lines divide the process in three different phases: in the *modeling* phase, a (annotated[1]) software model is built; in the *performance analysis* phase, a performance model is obtained through model transformation, and such model is solved to obtain the performance results of interest; in the *refactoring* phase, the performance results are interpreted and, if necessary, feedback is generated as refactoring actions on the original software model.

Approaches aimed at identifying and removing the causes of poor performance in software systems commonly fall in two categories: (i) *bottleneck analysis* that allows to identify cases when the performance of a software system are limited by a number of overloaded software components and/or hardware resources [11]; (ii) *performance antipatterns* that document common mistakes made during software development, as well as their solutions [25, 8].

[1]Annotations are aimed at specifying system parameters such as workload, service demands and hardware characteristics.

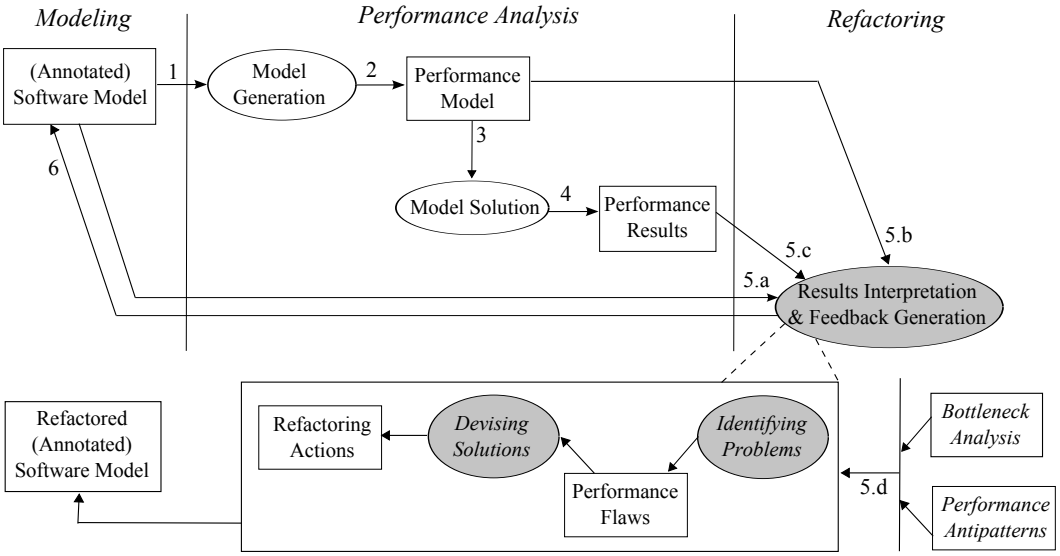


Figure 1: Exploiting bottleneck analysis and performance antipatterns in the round-trip SPE process.

In Figure 1 the (annotated) software model (label 5.a), the performance model (label 5.b), and the performance results (label 5.c) are all inputs to the *results interpretation & feedback generation* step that searches problems in the model. This step has been expanded in the bottommost part of the figure, where a fourth input has been added, that is the two analysis techniques (label 5.d) we consider in our approach, i.e. *bottleneck analysis* and/or *performance antipatterns*. In general, the performance analysis results have to be interpreted in order to identify, if any, performance problems. Once *performance flaws* have been identified (with a certain accuracy) somewhere in the software model, it is necessary to devise solutions in terms of *refactoring actions* that have to be applied to remove those flaws. A performance flaw originates from a set of unfulfilled requirement(s), such as "the estimated average response time of a service is higher than the required one". If all the requirements are satisfied then the feedback obviously suggests no changes. Both considered approaches follow the same general process but they relay on different instruments.

The goal of this paper is to look for possible synergies between performance antipatterns and bottleneck analysis, in order to strengthen the feedback process by providing to designers a sufficiently large set of alternatives for improving the system performance.

The remainder of the paper is organized as follows. Section 2 presents related work; Section 3 describes our approach; Section 4 shows the approach at work on a case study from the e-commerce domain; Section 5 reports the lessons learned from the experimentation as well as the open issues raised by the approach; and finally Section 6 concludes the paper and provides directions for future research.

2. RELATED WORK

The work presented in this paper is related to two main research areas and builds upon our previous results in these areas: (i) bottleneck analysis, and (ii) performance antipatterns.

Bottleneck analysis. In [29] an approach has been presented for automated software performance diagnosis, which identifies performance flaws before the software system implementation. It defines a set of *rules* for detecting patterns of interaction between resources. The software architectural models are translated into a performance model, i.e. Layered Queueing Networks (LQNs) [21], [28] and then analyzed. The approach limits the detection to bottlenecks and long execution paths identified and removed at the level of the LQN performance model. The overall approach was applied only to LQN models, so its portability to other notations is yet to be proven.

In [14] we studied the impact of SOA design patterns on the performance analysis of Service Oriented Architectures (SOA), and in [15] we described a technique for automatically refactoring a SOA design model by applying a SOA design pattern and then propagating the incremental changes to its LQN performance model.

Performance antipatterns. Enterprise technologies and EJB performance antipatterns are analyzed in [20]: antipatterns are represented as sets of rules loaded into an engine. A rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However, it deals with Component-Based Enterprise Systems, targeting only Enterprise Java Bean (EJB) applications, hence its scope is restricted to such domain, and performance problems can neither be detected in other technology contexts nor in the early development stages.

In [4] we have introduced an approach based on a role-modelling language that allows the refactoring of software models through removing performance antipatterns, and in [3] we used model-driven techniques, i.e. model differencing [6], to automatically refactor software models by applying performance antipatterns.

In the general context of software model optimization methods, which aim to automate the search for an optimal design with respect to a (set of) quality attribute(s), a considerable amount of work has been based on strategy techniques aimed at exploring different degrees of freedom (e.g., allocation, sw/hw replication and/or selection, etc.) that influence the system quality [2].

In the area of software design quality improvement, several search-based refactoring techniques have been proposed. In [22], a search-based approach for refactoring the class structure of a software system is proposed, but it is limited to a restricted set of refactorings. In [12], search-based techniques are used to automatically discover useful refactorings aimed at improving the quality of software systems. Authors use the concept of Pareto optimality to search-based refactoring, hence multiple fitness functions lead to provide different Pareto optimal refactorings. In [19], multiple weighted metrics are combined into a single fitness function that is based on well-known measures of coupling between program components. All these search-based approaches share the same limitation, i.e., the search space may be huge, so the search process may be time-consuming. In the performance domain, Koziolek et al. in [16] used meta-heuristic search techniques for improving performance, reliability, and costs of component-based software systems. In particular, evolutionary algorithms search the architectural design space for optimal trade-offs by means of Pareto curves.

To summarize, this is the first paper, to the best of our knowledge, that combines two different and well-consolidated analysis techniques for producing feedback to designers on how to improve the system performance.

3. SYNERGY ANALYSIS PROCESS

In this section we present the process we follow to explore the synergies between the Bottleneck Analysis (BA) [11] and the Performance Antipatterns (PA)[8].

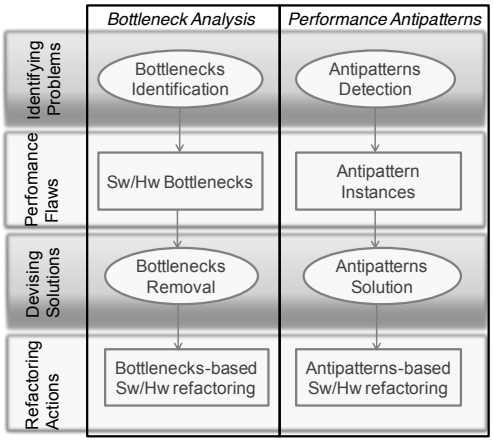


Figure 2: Customizing the refactoring phase.

Figure 2 specializes the general *Results Interpretation & Feedback Generation* process of Figure 1 in case either the BA or the PA are used to interpret the performance analysis results and to generate feedback on the software model. We recall that feedback is aimed at improving the software system performance in order to reach the goal of fulfilling the performance requirements.

The **BA** aims to identify and remove the system bottleneck. More specifically, by system bottlenecks we understand one (or a small number) of software or hardware resources that are highly utilized and will be the first to saturate, throttling the system performance. The system bottleneck indicates an imbalance in the use of resources, which needs to be resolved in order to fully utilize all the resources in the system. Hence in the BA, problem identification corresponds to *Bottlenecks Identification* which determines the *Sw/Hw bottleneck* present in the system by looking at the performance utilization of software components and hardware platforms. The devising solution step corresponds to *Bottleneck Removal* that returns the list of *Bottlenecks-based Sw/Hw refactoring* actions to be applied to the initial system model in order to improve the performance. Examples of such refactoring actions are setting the multi-threading configuration for software components and the multi-processor configuration for hardware platforms, or re-allocating the work among the system resources [29].

The **PA**, instead, aims to detect and remove performance antipatterns introduced in the software system during the design. A performance antipattern [25] identifies a problem, i.e. a bad design practice that negatively affects the software performance, and a solution, i.e. a set of refactoring actions that should be applied to remove it. Hence in the PA, problems identification corresponds to *Antipatterns Detection* step that, looking at the software models and the performance indices, identifies a list of *Antipattern Instances*. The *Antipatterns Solution* step suggests a set of *Antipatterns-based Sw/Hw refactoring* actions to obtain a new software system with improved performance. In PA, refactoring actions span from redesign software components in terms of internal behavior or their external communication, set multi-treading configuration for software components, to redeployment strategies. Note that the solution of one or more antipatterns does not a priori guarantee performance improvements, because the entire process is based on heuristic evaluations [9]. However, an antipattern-based refactoring action is basically a correctness-preserving transformation that aims at improving the quality of the software.

In this paper, we introduce an analysis process to explore possible synergies between PA and BA. Such process includes the following options:

1. Execute BA and PA separately. We compare their output results in terms of what are the refactoring actions the two techniques propose and the performance improvements we get by applying such actions. In this way it is possible to provide evidence of the relative strengths and weaknesses of the two methods.

2. Execute BA and PA alternatively. We merge the two techniques: (i) if BA is executed first, the system bottleneck will be alleviated, reaching a system configuration where there is no obvious imbalance in the usage of resources; however it is possible that the performance requirements are still not fulfilled, hence PA may be useful to further improve the output of BA; (ii) if PA is executed first, there are no bad design practices in the software system, however it may happen that it still includes sw/hw bottlenecks that throttle the system performance, hence BA may be useful to further improve the output of PA.

3. Reduce the PA solution space by means of BA. We use the output of BA *bottleneck identification* step to reduce the PA solution space by pruning the graph of design alternatives (i.e., solve the antipatterns that involve bottlenecks exclusively) thus to quickly converge towards a refactored software model that, even if it is not the optimal one, shows better performance and possibly satisfies the stated requirements.

The goal of our synergy analysis process is to strengthen the *Results Interpretation & Feedback Generation* step (see Figure 1) by increasing the performance improvement capabilities. In particular, the combination of BA and PA offers a powerful support to software engineers, since it provides a broader sets of design alternatives that may include specific solutions leading to better performance results.

4. CASE STUDY

The proposed approach is illustrated by means of a case study from the e-commerce domain, which has been modeled using UML [1]. Figure 3 shows the Use Case Diagram of the E-Commerce System (ECS). It is a web-based system that manages business data related to books and movies: *guest* users can browse catalogues and, at the same time, *customer* users can make selections of items that need to be purchased.

Software model annotations have been defined to support the performance analysis. Figure 3 uses MARTE [18] annotations to specify the system workload. In particular: (i) a closed workload has been defined for the *BrowseCatalog* service, for which the number of users is set to 98 with an average thinking time of 3 seconds; (ii) a closed workload has been defined for the *MakePurchase* service, with a population of 2 users with an average thinking time of 5 seconds.

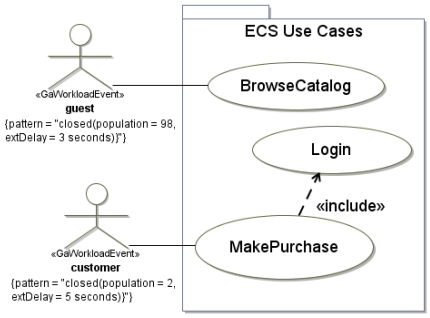


Figure 3: ECS- Use Case Diagram.

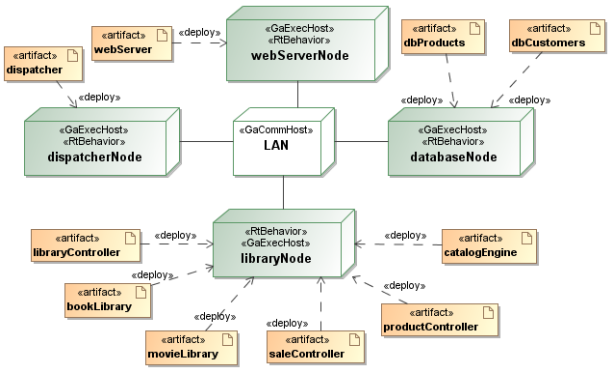


Figure 5: ECS- Deployment Diagram.

The UML Component Diagram shown in Figure 4 describes the software components and their dependencies. *guestApp* and *customerApp* components are connected to the *webServer* component that forwards users' requests to the *dispatcher* component. This latter component forwards the requests related to the *browseCatalog* service towards the *libraryController* whereas requests related to the *makePurchase* service are handled by the *saleController* component. *bookLibrary* and *movieLibrary* components manage books and movies, respectively, by invoking the *catalogEngine* component that retrieves information from the *dbProducts* component. The purchases are in charge of the *productController* that communicates with *dbCustomers* and *dbProducts* to retrieve the information to successfully accomplish the purchase. The UML Deployment Diagram depicted in Figure 5 shows the deployment of software artifacts onto hardware devices (i.e., *webServerNode*, *dispatcherNode*, *libraryNode*, and *DatabaseNode*) communicating through a Local Area Network (LAN).

Note that we consider as starting ECS configuration for the analysis (called *base case* in the rest of the paper) the ECS case with single-threaded hardware and multi-threaded software (30 threads for the *WebServer* software component, and 20 threads for all the other software components, except the *dispatcher*). This configuration comes from the BA evaluation of ECS and it represents a necessary premise to the following analysis since it provides an appropriate concurrency level thus to avoid undesirable situations of software bottleneck, where all hardware resources are under-utilized. We considered that a more realistic operating point of the system would allow for full utilization of at least one hardware resource. Please refer to Section 4.1 for more details on the BA evaluation.

The performance requirements imposed on the *BrowseCatalog* and *MakePurchase* services are: (ii) the average response time of the *BrowseCatalog* service must not exceed 4 seconds. (ii) the average response time of the *MakePurchase* service must not exceed 8 seconds. Both requirements need to be fulfilled under the closed workloads defined for the *guest* and *customer* users, respectively.

The performance analysis has been conducted by transforming the software model into a Layered Queueing Network (LQN) model [28], shown in Figure 6, and solving it with the LQN Solver tool [10].

Requirement	Required Value	Predicted Value
RT(*BrowseCatalog*)	4 sec	7.73 sec
RT(*MakePurchase*)	8 sec	91.99 sec

Table 1: Response time of the ECS software model.

Table 1 reports the response times of the ECS base case model. First column reports the required index, second column the required value, and third column the predicted value (obtained from the LQN analysis). As it can be noticed, both services have response times exceeding the required ones, hence a deep analysis must be conducted to identify performance flaws and to devise solutions improving such indices. In the following we first apply the bottleneck (see Section 4.1) and the performance antipatterns (see Section 4.2) analysis techniques, then we combine these techniques in Section 4.3 to explore their synergies.

4.1 Bottleneck Analysis

The performance analysis results of the ECS system for the default configuration of single-threaded software components and hardware platforms shows a strong case of software bottleneck under the defined closed workload. An undesirable effect of software bottleneck is that none of the hardware resources gets to be utilized at full capacity, thus wasting costly system resources. Software bottleneck can be resolved by increasing the number of threads of the sw components, which raises the concurrency level in the software

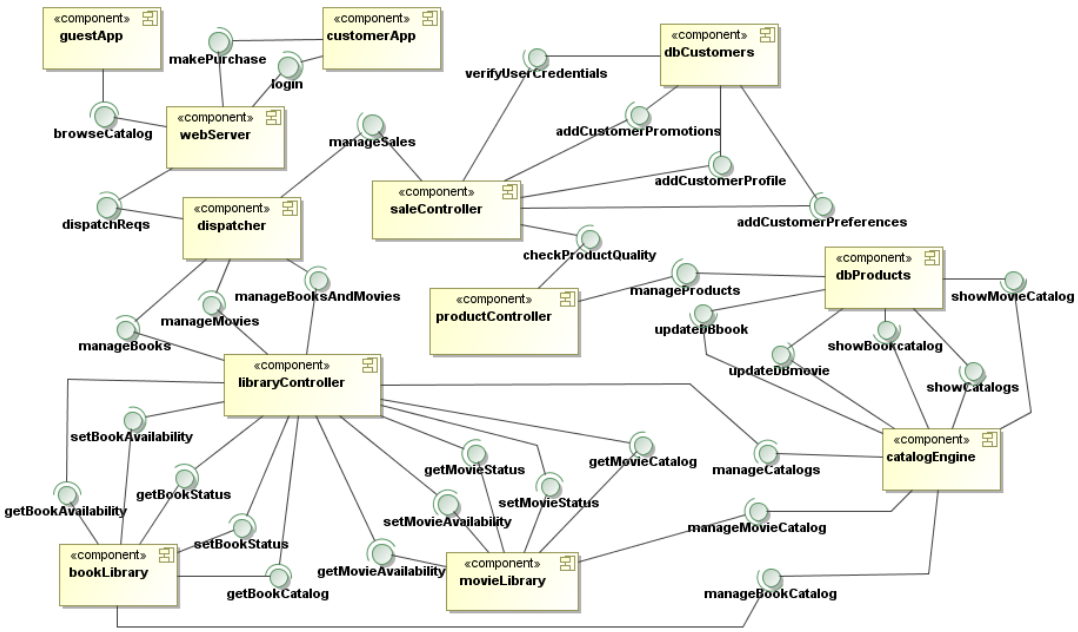


Figure 4: ECS- Component Diagram.

and pushes more workload to the hardware. A first set of experiments showed that a system configuration of 30-thread instances for the *webServer* component and 20-thread the remaining software components (with the exception of the *dispatcher*, which is single-threaded) moves the system bottleneck from software to hardware. The response time for the two classes of users (with 98 and 2 users, respectively) are 7.73 sec for *browseCatalog* and 91.98 sec for *makePurchase*. Since multi-threading the software to avoid wasting the hardware resources is often used in practice, we consider the ECS configuration with multi-threaded software components (with 30 and 20 threads, as described above) running on single-instance hardware processors as the base case for applying the BA and PA analysis.

The next BA experiments aim to remove the hardware bottleneck from the base case. LQN results show that the *libraryNode* and *databaseNode* hardware platforms are both saturated (i.e., with utilizations of 0.98 and 0.92 respectively).

A commonly adopted solution for removing this type of hardware bottleneck is to increase the number of instances for the saturated processors. Therefore, we repeat the experiment by increasing the instances of those processors from 1 to 4 and observe their utilization. Figure 7 reports the utilization of the processors under study, which decreases when the number of processor instances increases. Such refactoring actions imply faster response time for both services as shown in Figure 8. Figure 7 reports the utilization of *libraryNode* and *databaseNode* processors, which is decreasing when the number of processor instances increases. Such refactoring actions have as effect a faster response time for both services as shown in Figure 8. In particular, for the case of 4 processor instances, the response times of the two classes has improved from 7.73 to 3.76 sec. for *BrowseCatolog* service, and from 91.99 to 30.37 sec. for *MakePurchase* service. Actually, the hardware bottleneck has been removed already with 3 processor instances, but since 4-core processors are very common, we select 4 as the suggested solution.

4.2 Performance Antipatterns

Table 2 reports the output of the antipatterns detection [8]: six instances of different antipatterns have been found, i.e., Circuitous Treasure Hunt (CTH), Concurrent Processing Systems (CPS), Blob, Extensive Processing (EP), Empty Semi Trucks (EST). For example, the CTH antipattern occurs since the *saleController* component needs to invoke the *dbCustomers* database component several times before providing the user *Login* service.

Antipattern	Problem
CTH	The *saleController* component needs to invoke the *dbCustomers* database component several times before providing the user *Login* service.
CPS_x	The *databaseNode* hardware platform is much more utilized than *dispatcherNode*.
$BLOB$	The *libraryController* component performs most of the work and an excessive number of messages are exchanged with *bookLibrary* and *movieLibrary* components.
EP	The *catalogEngine* component requires extensive processing to *manageCatalogs* in comparison to *manageBookCatalog* and *manageMovieCatalog* separately.
EST	The *saleController* component needs to invoke the *productController* component several times before providing the *checkProductQuality* service.
CPS_y	The *libraryNode* hardware platform is much more utilized than *dispatcherNode*.

Table 2: ECS - detection of antipatterns.

Table 3 reports the refactoring actions we applied to solve the detected performance antipatterns. For example, the CTH antipattern is solved by refactoring the communication between *saleController* and *dbCustomers* thus to avoid an excessive number of messages.

Note that the detected antipatterns affect different software model entities hence their solution can be incrementally combined without incurring in conflicting and divergent refactorings.

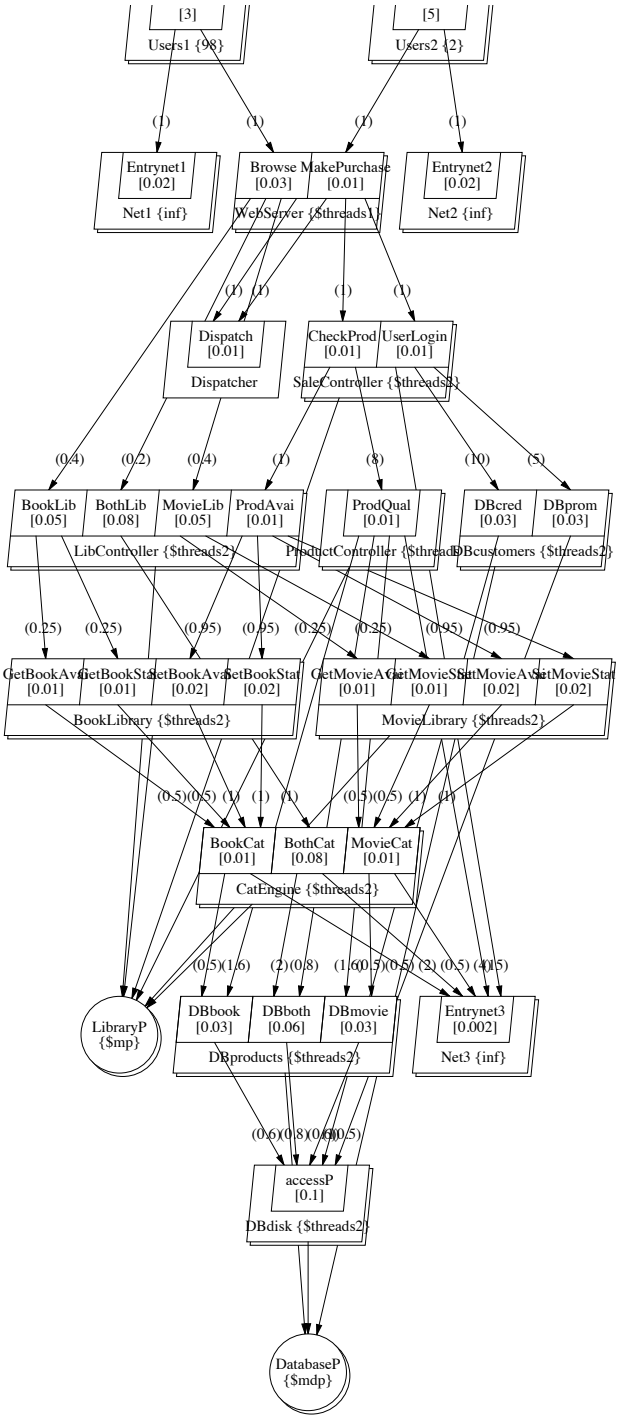


Figure 6: ECS- performance model.

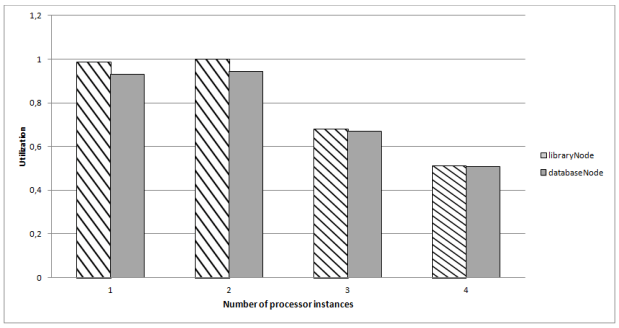


Figure 7: ECS- Utilization of hardware platforms while increasing the number of processor instances.

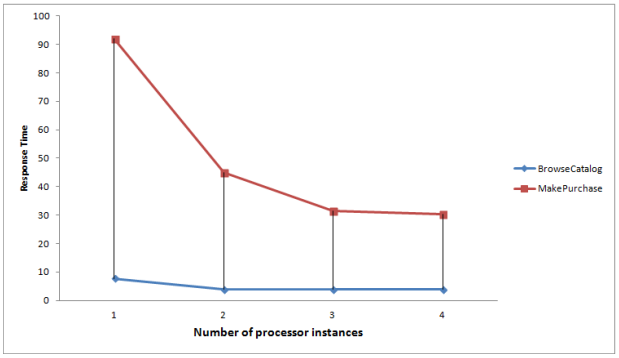


Figure 8: ECS- Response time of software services while increasing the number of processor instances.

Several iterations can be conducted to find the software model that best fits the performance requirements, since several antipatterns have been detected in the software model. At each iteration, the refactoring actions suggested by one antipattern produce a new software system design that replaces the analyzed one. Then, the detection and solution approach can be iteratively applied to all newly generated candidates to further improve the system.

Figure 9 reports the output of the antipatterns-based approach [9]. It is a graph where each node represents a design alternative and each arc is labeled with the name of the antipattern that has been applied to refactor the software model (see more details in Table 3). The ECS base case is labeled *0* and represents the root of the graph. The remaining nodes are labeled by means of digits representing the removed antipatterns, following the antipattern-to-digit mapping indicated by the legend in the bottom of Figure 9. For example, the node labeled *2.3.5* represents the ECS system where CPS_x (i.e., 2), $BLOB$ (i.e., 3) and EST (i.e., 5) antipatterns have been solved. We recall that the solution order of antipatterns is invariant since the detected antipatterns affect different software model entities, hence the node *2.3.5* is equivalent to all nodes represented by other permutations of the three digits (e.g., node *2.5.3*) that we intentionally hide in Figure 9.

Each node reports the response time of *BrowseCatalog* and *MakePurchase* services that for sake of figure readability we name *rBC* and *rMP*, respectively. Note that the solution of one or more antipatterns does not guarantee performance improvements in advance: the entire process is based on

Antipattern	Solution
CTH	The communication between *saleController* and *dbCustomers* has been refactored to avoid an excessive number of messages.
CPS_x	The *dbCustomers* component has been deployed from *databaseNode* to *dispatcherNode* in order to optimize the usage of available hardware resources.
$BLOB$	The communication between *libraryController* and *bookLibrary movieLibrary* components has been refactored by delegating some work to these latter components.
EP	The extensive processing has been delegated to a mirrored component of *catalogEngine*, called *catalogEngineMirror*, whereas the processing of *manageBookCatalog* and *manageMovieCatalog* components is still handled by the *catalogEngine*.
EST	The communication between *saleController* and *productController* has been refactored to avoid an excessive number of messages.
CPS_y	The *saleController* component has been deployed from *libraryNode* to *dispatcherNode* in order to optimize the usage of available hardware resources.

Table 3: ECS - solution of antipatterns.

heuristics evaluations. For example, if we compare the node labeled *1.2* with the node labeled *1.2.3* we can notice that this latter node improves the first index (i.e. the response time of the *BrowseCatalog* service varies from 7.75 to 6.54 seconds) but it makes worse the other index (i.e. the response time of the *MakePurchase* service varies from 68.39 to 80.63 seconds).

To compare different design alternatives and to identify the best one, we weight them using the metrics:

$$rBC * p_1 + rMP * p_2 \quad (1)$$

where p_1 and p_2 represents the priority of rBC and rMP requirements respectively.

In our case study they are equally weighted to 0.5, and the best design alternative corresponds to the lowest weight that is achieved with the node labeled *1.2.3.5.6* where RT(BrowseCatalog)= 6.4 sec and RT(MakePurchase)= 19.58 sec.

The node labeled *1.2.3.5.6* corresponds to a design alternative where CTH (i.e., 1), CPS_x (i.e., 2), $BLOB$ (i.e., 3), EST (i.e., 5), and CPS_y (i.e., 6) antipatterns have been solved. PA gives as output a refactored software model that includes the following refactoring actions:

1. the communication between *saleController* and *dbCustomers* has been refactored by avoiding an excessive exchange of messages and moving the computation from *saleController* to *dbCustomers*. In particular, the *Login* service was performed by invoking 10+5 times the *dbCustomers* component. In the refactoring the computation is moved to the *dbCustomers* component, hence the *Login* service is performed by invoking the *dbCustomers* once to check users credentials (whose demand increases from 0.03 to 0.09) and once to verify customer promotions (whose demand increases from 0.03 to 0.06);
2. the *dbCustomers* component has been redeployed from *databaseNode* to *dispatcherNode*;
3. the communication between *libraryController* and *bookLibrary movieLibrary* components has been refactored. In particular, the *BrowseCatalog* service was performed by concentrating the business logic in the *libraryController* and invoking the *get* and *set* operations only of *bookLibrary* and *movieLibrary*. In the refactoring these latter components have been redesigned and the computation is moved from *libraryController* (whose demand decreases from 0.05 to 0.02) to *bookLibrary* and *movieLibrary* (whose demands increase from 0.03 to 0.045);
4. the communication between *saleController* and *productController* has been refactored, as shown in Figure 10. In particular, the *checkProductQuality* service was performed by invoking 8 times the *productController* component. In the refactoring the computation is moved to this latter component, hence the service is performed by invoking once such component to check the quality of products (whose demand increases from 0.01 to 0.03);
5. the *saleController* component has been redeployed from *libraryNode* to *dispatcherNode*.

All these refactoring actions have been applied on the ECS initial system. Figure 11 reports the LQN performance model corresponding to the refactored software model, where all the performance parameters (i.e., tasks and processors information as well as the frequency of calling entries) have been visualized.

4.3 Identifying synergies between BA and PA

In this Section, we first analyze the results of BA and PA executed separately on ECS in order to point out the strengthens and weakness of both techniques (see Section 4.3.1). Then, being guided by the ECS experimentation, we discuss the synergies between BA and PA, that try to overcome the identified limits. In particular, we envisage two types of synergies: (i) *combination of the two techniques*, i.e., one technique is executed on the results obtained by the other one (see Section 4.3.2); (ii) *pruning the PA graph via BA*, i.e., the results of BA are used to reduce the PA solution space to quickly converge to a design alternative with better performance (see Section 4.3.3).

4.3.1 Execute BA and PA separately

BA provides a software model candidate that greatly improves the response time of the *BrowseCatalog* service (from 7.73 sec to 3.76 sec satisfying the corresponding requirement) but it does not fully benefit the response time of the *MakePurchase* service (from 91.99 sec to 30.37 sec). PA provides a software model candidate that slightly improves the response time of the *BrowseCatalog* service (from 7.73 sec to 6.4 sec) but it provides more benefit for the response time of the *MakePurchase* service (from 91.99 sec to 19.58 sec).

BA gives as output a refactored software model where the instances of *libraryNode* and *databaseNode* have been increased from 1 to 4. Such refactoring action suggests to potentiate two processors whose cost is affordable today.

PA gives as output a refactored software model where two main refactoring actions have been performed: (i) redeployment of software components, in fact the *dbCustomers* component has been redeployed from *databaseNode* to *dispatcherNode*, and the *saleController* component has been redeployed from *libraryNode* to *dispatcherNode*; (ii) software components and communication redesign to reduce communication latency between *saleController* and *dbCustomers*, between *libraryController* and *bookLibrary* and *movieLibrary*, between *saleController* and *productController*. While the cost of the first type of refactoring is quite low, the second one could be very expensive since it involves human work. Of course, the amount of these expenses depend on

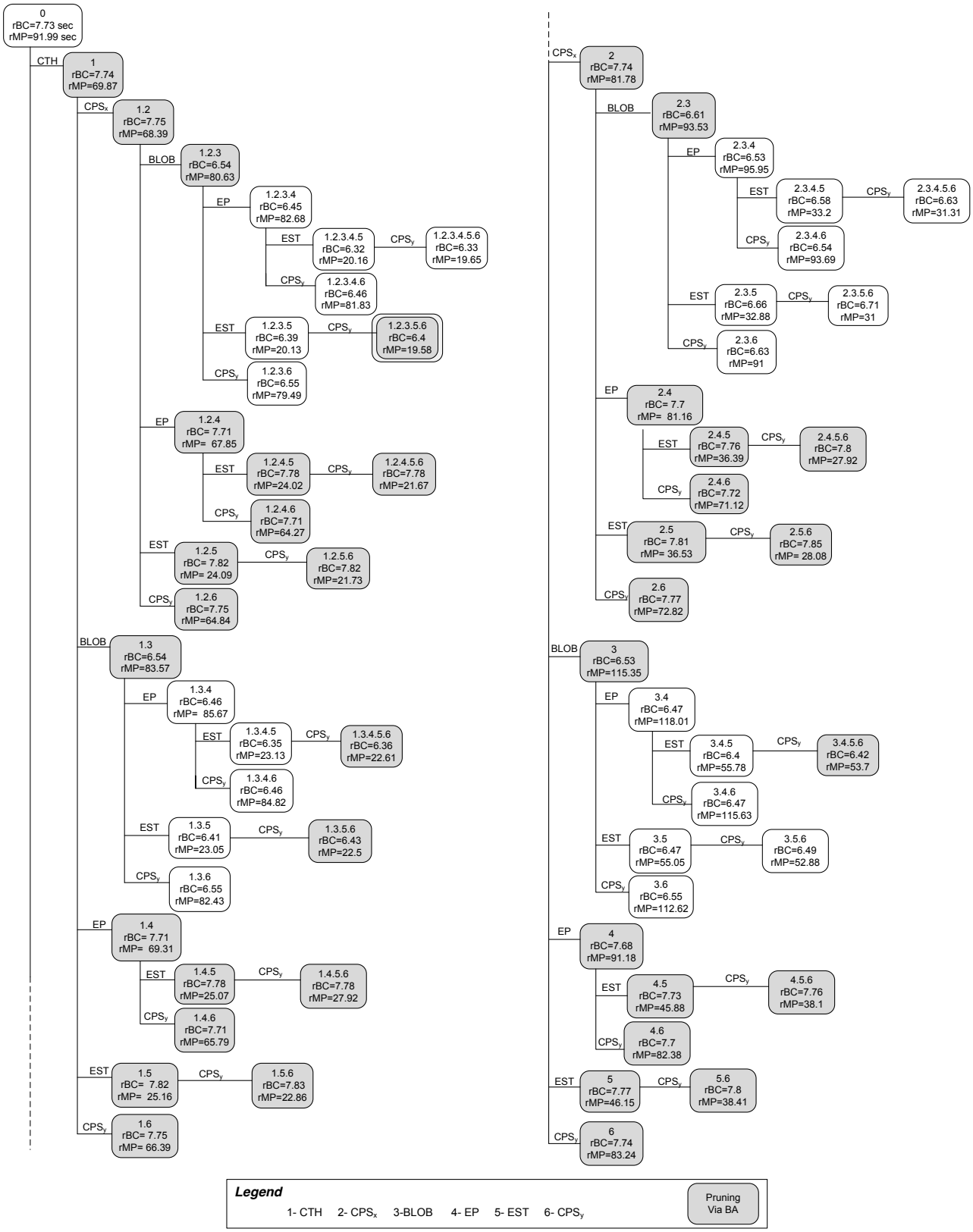
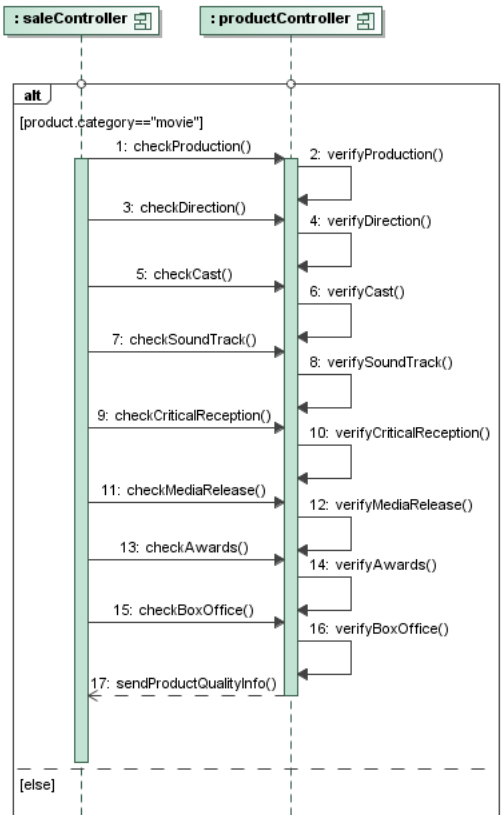
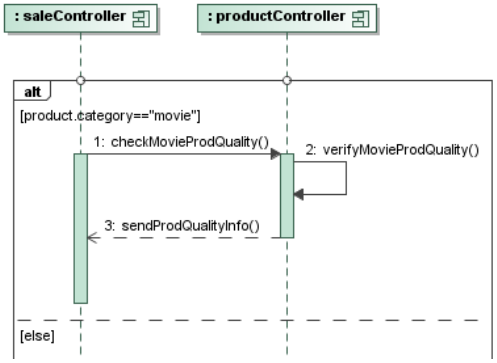


Figure 9: ECS - reduce the PA solution space by means of BA.



(a) An excerpt of ECS software model.



(b) An excerpt of ECS refactored software model.

Figure 10: ECS- solving the *EST* performance antipattern.

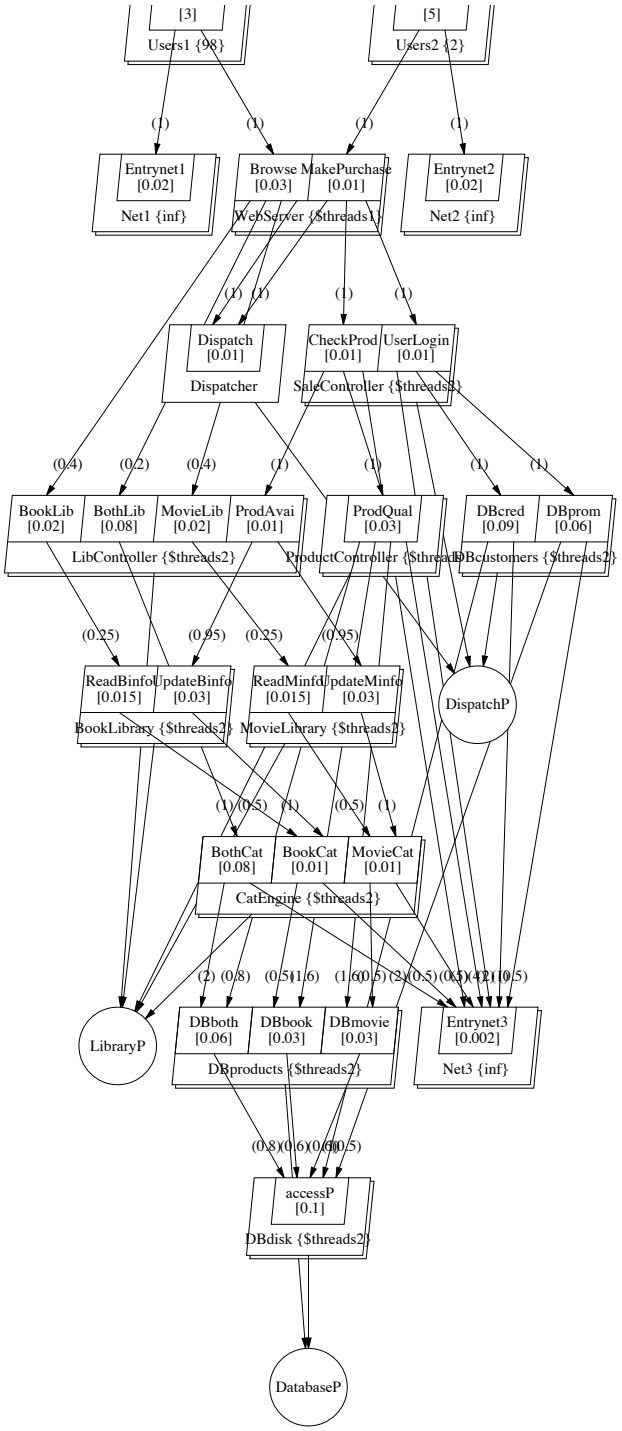


Figure 11: ECS - refactored performance model.

the complexity of the software system, on the complexity of the changes and on impact of them on the whole system.

To compare the goodness of the PA and BA suggested design alternatives, we weight the design alternatives by using the metrics (1) introduced in Section 4.2, where p_1 and p_2 are equally set to 0.5. Thus, the ECS base case is weighted 49.86, whereas the ECS after BA is weighted 17.06 and ECS after PA is weighted 12.99. The benefit of these techniques w.r.t. performance improvements is estimated by comparing the percentage of improvement achieved in this weighted sum, hence BA brings a benefit of 65.77% whereas PA brings a benefit of 73.95%. Even if BA allows to satisfy one requirement, it performs slightly worse while considering the combination of both requirements. However, the refactoring actions suggested by BA are less expensive than the ones supported by PA, since the latter requires the redesign of several software components that may involve expensive human re-work.

As final consideration, both techniques in isolation fail to suggest an alternative satisfying the performance requirements.

4.3.2 Execute BA and PA alternatively

Figure 12 reports the results of executing BA and PA alternatively. In the figure, nodes represent design alternatives with the corresponding response time of both services, while arcs are labeled by the re-factoring actions executed to obtain the reaching nodes. The root of the graph is the ECS base case.

If we first execute BA and then PA (the left-hand path) we get a software model candidate that greatly improves the response time of the *BrowseCatalog* service (from 7.73 sec to 3.83 sec) but it does not fully benefit the response time of the *MakePurchase* service (from 91.99 sec to 24.88 sec). The suggested design alternative is the one described in Section 4.1 where only CPS_y antipattern has been detected and solved, redeploying the*saleController* component from *libraryNode* to *dispatcherNode*.

On the contrary, if we first execute PA and then BA (the right-hand path of Figure 12) we get a software model candidate that greatly improves the response time of the *BrowseCatalog* service (from 7.73 sec to 3.33 sec) and the response time of the *MakePurchase* service (from 91.99 sec to 6.56 sec), and that, indeed, fulfills both performance requirements (as indicated by the shaded box of Figure 12). The suggested design alternative is the one described in Section 4.2 where *libraryNode* and *databaseNode* are 4-core processors each.

Similarly to the estimation done to compare BA and PA separately, ECS after "*BA+PA*" is weighted with 14.35, and ECS after "*PA+BA*" is weighted with 4.94. The benefit of executing these techniques alternatively is estimated by comparing the percentage of improvement achieved in this weighted sum, hence *BA+PA* brings a benefit of 71.21% whereas *PA+BA* brings a benefit of 90.08% with respect to the initial ECS weighted with 49.86.

4.3.3 Reduce the PA solution space by means of BA

Another way to exploit the synergy between BA and PA is to reduce the PA solution space by means of BA, i.e., by pruning the graph of design alternatives using the knowledge coming from BA. The goal is to quickly get a "good enough" design alternative without building the whole graph of design alternatives. Of course, the strategy is an heuristics that might not bring to the best design alternative PA can identify, but towards an alternative that, even if it is not the optimal one, shows better performance and possibly satisfies the stated performance requirements.

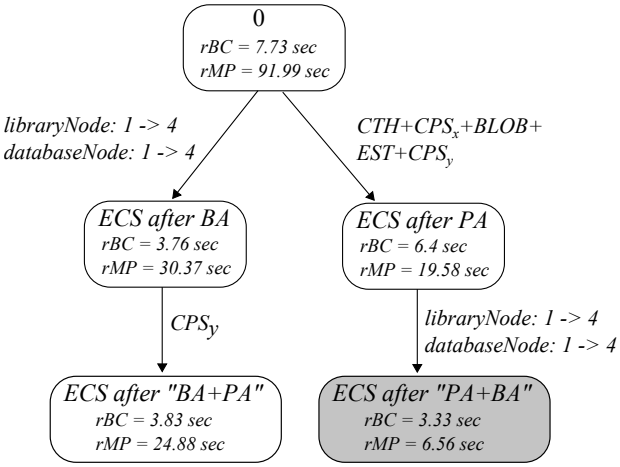


Figure 12: ECS- performance indices while executing BA and PA alternatively.

The pruning strategy we device suggests to keep all the nodes (of the alternative designs graph) obtained by removing antipatterns instances on hardware bottlenecks, and to discard all the others. In particular, we here consider hardware bottlenecks all the devices showing an utilization higher than 0.8.

In Figure 9 the result of the pruning strategy on the ECS case study is shown by indicating with the grey nodes the design alternatives we keep. The devised strategy allows to prune 24 nodes over 63 design alternatives the PA process builds for the ECS system (see Section 4.2 for more details), reaching the 38.1% percentage of pruning. Note that in our case study the BA heuristics allows to reach the optimal PA alternative, however this is not guaranteed in general and we intend to investigate this issue in the near future.

5. DISCUSSION

In this Section we discuss the lessons learned from the experimentation as well as the open issues raised by the approach.

Limitations of bottleneck analysis. BA is a technique that mitigates the bottleneck and balances the usage of resources. Once this goal is reached, BA cannot further help to improve performance, then PA should be used to get more insights on how to further improve the system performance. In order to better understand the ECS base case characteristics, we conducted a performance analysis without contention. The analysis reports that the lower bound for the response times of the *BrowseCatalog* and *MakePurchase* services are 3.20 sec and 12.70 sec, respectively. Again, the response time of *MakePurchase* service is far from the performance requirement (i.e. 8 sec), thus demonstrating that even in case of the best option (i.e., no contention), the system fails to satisfy the requirements. In order to improve it further, we need to change the design, e.g., by introducing some concurrency in the execution path of *MakePurchase* requests. This cannot be done with BA, whereas PA provides

more insights on possible refactoring actions that conduct to a better design. By detecting and removing the performance antipatterns we are able to redesign the system and, in our case study, we experience the best performance when merging the two analysis techniques. Indeed, only exploiting together BA and PA we reach an ECS design that satisfies both response time requirements.

Limitations of performance antipatterns. Our formalization of performance antipatterns [8] is based on a set of thresholds that, if not properly set, may hide bad design. Hence, the threshold tuning is a difficult task that may affect the accuracy of antipattern detection. Moreover, in this paper context the experimentation demonstrates that, if we firstly execute the bottleneck analysis and the relative refactoring actions, several performance antipatterns are hidden, as happened in ECS when PA is executed after BA (see Section 4.3.2) and only CPS_y has been detected and solved. In fact, while the bottleneck analysis is aimed at keeping the utilization of hardware devices and software tasks under certain thresholds, high utilization values are fundamental to detect many performance antipatterns [8]. If we apply PA detection on the system configuration provided by BA (i.e., the one discussed in Section 3), then most of the antipatterns are not identified, due to their limited sw/hw utilization.

Complexity vs Effectiveness. Performance antipatterns are very complex to detect because they are founded on different characteristics of a software system, spanning from static to behavioral to deployment, and they additionally include values of performance indices. However, this complexity subsumes a wide variety of refactoring actions to express, thus making this approach very powerful in the identification of performance flaws and system refactoring. Hence, the complexity is rewarded by expressiveness. As opposed, bottleneck analysis is a well-assessed technique widely supported by a solid theory and sophisticated tools. Hence, the detection of bottlenecks in performance models is not such a complex task in general. The cost to pay to this reduced complexity, as outlined above, is the limitation in expressiveness of repairing actions. BA is particularly powerful in case of good system design when the performance problems come from unbalanced load or under-estimated resources. On the contrary, it cannot help in case the performance flaw originates from software system development. PA, instead, should be applied when performance problems come from design choices and software system re-design is necessary. In fact, it gives insight on what happens in the software model and suggests solutions for modifying it. Our experimentation demonstrates that there are cases where an unsatisfied requirement cannot be overcome by only adding hardware resources, since there is a problem in software design. In these cases, there is a point beyond which if we add more hw/sw resources we do not gain better performance, or even the performance worsen. For example, the ECS base case with 4 processor instances has no more bottlenecks, but the response time for *MakePurchase* is far from satisfying the requirement. Summing up, it is preferable to execute BA first and, in case of specific constraints on the resources or in case of unsatisfactory requirements, to proceed with PA, while taking into account that BA can hide key performance antipatterns as happened in our experimentation.

Cost/Effort issues. PA costs derive to performance antipattern detection and solution complexity, that is the counterpart of their expressiveness and wide impact on the whole system design. BA costs are instead more related to the skills and experience of performance analysts. In our case, we had to solve about 13 LQN models, while continuously changing/tuning model parameters, before removing software/hardware bottlenecks. Hence, we think that quantifying the effort required to apply BA, PA, or their combination is very difficult since both techniques have several limitations and (complexity, cost) issues cannot be avoided. Such estimation has to take into account some factors, that we intend to further investigate, such as: (i) the degree of automation, (ii) the design/performance skills required to achieve the design alternatives, (ii) the scalability in terms of number of analysed performance models together with their complexity and performance gain.

BA and PA synergies. The experimentations on ECS show that several synergies can be exploited to improve performance or to reduce the size of PA solution space. One synergy consists in alternating PA and BA. The combined usage of both techniques permits to make a step ahead, and in particular the order PA before BA is the only strategy that, on the ECS case study, conducts to a system design that satisfies both performance requirements (see Section 4.3.2). This result cannot be reached either executing separately the two analysis techniques or BA before PA, and it is justified by the fact that ECS base case suffers of bad design that throttles its performance. A second synergy has allowed us to define a heuristics based on BA that prunes the PA design alternatives graph. In this case, the BA output suggests, time by time, which antipattern instances have to be resolved and which ones can be discarded. For example, in ECS this heuristics has permitted to prune 38.1% of candidates (i.e., 39 LQN models have been solved over the 63 generated ones by PA), thus reaching the best design alternative of the whole graph by considerably reducing the costs of the PA detection and solution steps. The reduction of the PA solution space allows to speed-up the performance analysis. However, the duration of executing the performance solvers in the BA and PA may significantly vary on the basis of other application-dependent parameters (e.g., number of software and/or hardware resources) that indirectly affect the two analysis techniques.

6. CONCLUSION

This paper explores the synergies between Bottleneck Analysis and Performance Antipatterns techniques in the round-trip Software Performance Engineering (SPE) process. In order to identify strengths and weaknesses of both techniques, they have been separately applied to a software system in the e-commerce domain, and two types of synergies have been envisaged and experimented. The combination of these two techniques seems very promising, in fact we found that executing first the performance antipatterns and then the bottleneck analysis allowed to identify design alternatives satisfying all the performance requirements.

As future work, we intend to apply our approach to other case studies, possibly coming from real world systems. This wider experimentation will allow us to deeply investigate the effectiveness of BA heuristics that reduce the PA solution space, thus studying the scalability of our approach.

7. ACKNOWLEDGMENTS

This work was partially supported by the European Office of Aerospace Research and Development (EOARD), Grant

Cooperative Agreement (Award no. FA8655-11-1-3055), and the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery Grant program.

8. REFERENCES

[1] UML 2.0 Superstructure Specification, OMG document formal/05-07-04, 2005.

[2] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[3] D. Arcelli, V. Cortellessa, and D. Di Ruscio. Applying model differences to automate performance-driven refactoring of software models. In *European Workshop on Computer Performance Engineering (EPEW)*, pages 312–324, 2013.

[4] D. Arcelli, V. Cortellessa, and C. Trubiani. Antipattern-based model refactoring for software performance improvement. In *International ACM SIGSOFT conference on Quality of Software Architectures (QoSA)*, pages 33–42, 2012.

[5] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[6] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[7] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[8] V. Cortellessa, A. Di Marco, and C. Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Journal of Software and Systems Modeling*, 2012. DOI: 10.1007/s10270-012-0246-z.

[9] V. Cortellessa, A. Di Marco, and C. Trubiani. Software performance antipatterns: Modeling and analysis. In *Formal Methods for Model-Driven Engineering (SFM)*, pages 290–335, 2012.

[10] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. Layered Queueing Network Solver and Simulator, 2013. [online]http://www.sce.carleton.ca/rads/lqns/LQNS-UserMan-jan13.pdf.

[11] G. Franks, D. C. Petriu, C. M. Woodside, J. Xu, and P. Tregunno. Layered bottlenecks and their mitigation. In *International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 103–114, 2006.

[12] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *Conference on Genetic and evolutionary computation (GECCO)*, pages 1106–1113, 2007.

[13] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, 2010.

[14] N. Mani, D. C. Petriu, and C. M. Woodside. Studying the impact of design patterns on the performance analysis of service oriented architecture. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 12–19, 2011.

[15] N. Mani, D. C. Petriu, and C. M. Woodside. Propagation of incremental changes to performance model due to soa design pattern application. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 89–100, 2013.

[16] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *WOSP/SIPEW International Conference on Performance Engineering*, pages 105–116, 2010.

[17] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[18] Object Management Group (OMG). UML Profile for MARTE, 2009. OMG Document formal/08-06-09.

[19] M. O'Keeffe and M. í Cinnéide. Search-based refactoring for software maintenance. *J. Syst. Softw.*, 81(4):502–516, Apr. 2008.

[20] T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, 7(3):55–91, 2008.

[21] D. C. Petriu and H. Shen. Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In *Computer Performance Evaluation / TOOLS*, pages 159–177, 2002.

[22] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and evolutionary computation (GECCO)*, pages 1909–1916, 2006.

[23] C. U. Smith. Introduction to software performance engineering: Origins and outstanding problems. In *Formal Methods for Performance Evaluation, International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, pages 395–428, 2007.

[24] C. U. Smith and C. V. Millsap. Software performance engineering for oracle applications: Measurements and models. In *International Computer Measurement Group (CMG) Conference*, pages 331–342, 2008.

[25] C. U. Smith and L. G. Williams. More new software antipatterns: Even more ways to shoot yourself in the foot. In *International Computer Measurement Group (CMG) Conference*, pages 717–725, 2003.

[26] L. G. Williams and C. U. Smith. Software performance engineering: A tutorial introduction. In *International Computer Measurement Group (CMG) Conference*, pages 387–398, 2007.

[27] C. M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *International Workshop on the Future of Software Engineering (FOSE)*, pages 171–187, 2007.

[28] M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, and M. Alhaj. Transformation challenges: from software models to performance models. *Journal of Software and Systems Modeling*, 2013. accepted.

[29] J. Xu. Rule-based automatic software performance diagnosis and improvement. *Perform. Eval.*, 67(8):585–611, 2010.