Automatic, Model-Based Software Performance Improvement for Component-based Software Designs

Anne Martens [1]

Chair for Software Design and Quality
Universität Karlsruhe (TH) / KIT
76131 Karlsruhe, Germany

Heiko Koziolek[2]

ABB Corporate Research,
68526 Ladenburg, Germany

Abstract

Formal performance prediction methods, based on queueing network models, allow evaluating software architectural designs for performance. Existing methods provide prediction results such as response times and throughputs, but do not guide the software architect on how to improve the design. We propose a novel approach to optimise the expected performance of component-based software designs by automatically generating and evaluating design alternatives. The design space spanned by different design options (e.g. available components and configuration options) is systematically explored using metaheuristic search techniques and performance-domain heuristics. The gap between applying formal performance predictions and actually improving the design of a system can thus be closed. This paper presents a formal description and a prototypical implementation of our approach with a proof-of-concept case study.

Keywords: Component-based Software Engineering, Software Performance Engineering, Performance Prediction, Metaheuristics, Search-based Software Engineering, Design Space Exploration.

1 Introduction

Performance problems are continuously prevalent in many software systems [16]. Model-based prediction methods [1] try to tackle these problems during early design phases to avoid the problem of implementing architectures which are not able to fulfil certain performance goals. Based on architectural models of the system (e.g. UML models), the software architect can create formal analysis models (e.g. queueing networks [9]) to predict performance metrics such as mean response time or throughput.

[1] Email: martens@ipd.uka.de

[2] Email: heiko.koziolek@de.abb.com

However, most existing approaches do not provide further help after recognising that performance requirements are not met. The software architect currently needs to map the results back in the design model and then find new alternatives to the current design (e.g. by changing the selection of components, the configuration of components and containers, the sizing of resources) manually. It is hard to quantify the effect or choose the best from a set of design alternatives [12]. Some approaches [18,5] use heuristic rules, stating for example to increase the processing speed of bottleneck resources, to help the software architect. Mostly, however, these approaches are not automated. Additionally, the rules can only make use of the performance domain knowledge actually codified in these heuristics and cannot explore regions of the design space for which no prior knowledge exist. For example, the design choice for one of two available functional-equivalent components for a certain task can not be made in general, because the performance impact of the components depends on the specific system and its usage.

Additionally, the isolated improvements of performance properties alone is problematic, because changes to improve performance usually affect other properties of the system, e.g. cost or maintainability. Decisions to change the software architecture must take other extra-functional properties into account.

To meet the difficulty of solving detected performance problems, we propose an approach to automatically optimise the performance of a component-based software systems by automatically generating and evaluating design alternatives based on performance analyses of the software architecture. The novelty of our approach is the use of metaheuristic search techniques [6,8] (such as random-restart hill climbing, genetic algorithms or others) together with performance domain knowledge formalised as heuristics to systematically create and evaluate new architecture candidates. For example, bottleneck resources can be resolved by finding a better allocation of components to servers or changing the sizing of resources. In addition to solutions based on performance domain knowledge, we undirectedly (e.g. randomly) generate new candidates and integrate them in the search to allow a larger search space to be explored. In this work, we present a list of design change operations for the Palladio Component Model (PCM) [2] performance prediction approach. Additionally, we discuss how other extra-functional properties can be considered as constraints or even as additional decision criterion in a multicriteria optimisation in future versions of our approach.

In addition to the benefits of model-based performance prediction in general [1], the automated approach benefits software architects in two more ways: 1) The approach saves them time, as they do not have to explore the design space manually (by creating new candidates and evaluating them) and 2) the approach might result in better architectures than a manual exploration, as many more candidates can be evaluated in a shorter time.

The contributions of this paper are (i) an automated approach that applies metaheuristics using both performance domain knowledge and undirected, random search to improve the performance of component-based software systems on the model level and (ii) a proof-of-concept case study of a prototypical implementation of this approach. We aim to evolve the prototype into a framework for extra-functional property optimisation for component-based software systems.

The paper is structured as followed: Section 2 introduces the idea of optimisation of extra-functional properties in general using both heuristics and undirected search. Section 3 presents our approach which is specific to performance predictions for component-based systems and presents our current prototype implementation based on the PCM. In section 4, we present our proof-of-concept case study and explain the behaviour of the prototype step-by-step. Finally, section 6 presents the related work in the area of model-based software performance improvement, and section 7 concludes.

2 Extra-functional Property Optimisation on the Model Level

This section will first describe the optimisation problem for extra-functional properties (such as performance, reliability, or cost) on the model level formally (Section 2.1), before it sketches a generic, automated solution (Section 2.2).

2.1 Formal Description of the Design Space

We call the possibility to change a component-based system (including its allocation) in a certain way without affecting the functionality an (extra-functional) *design option*. For example, a component A in a design could be replaced by a number of components B, C and D that offer the same functionality, but have different performance characteristics. Also, resources offer design options, for example the number of replications of an application server or the processing speed of a CPU.

Let now I be a finite index set and D_i, $i \in I$ be our available design options and let $M = \{M_i\}, i \in I$ be the set of their value domains, which are assumed to be arbitrary, but countable sets. To continue with our examples, the value domain of a design option $D_{altComp}$: "replacing component A in the system" is $M_{altComp} = \{A, B, C, D\}$. The value domain of a design option $D_{CPU1speed}$: "change processing speed of CPU 1" is $M_{CPU1speed} = \mathbb{N}$ if measured in clock speed (e.g. in hertz).

The *design space* is defined as the Cartesian product $\mathcal{DS} = \Pi_{i \in I} M_i$. A *candidate* c is a tuple from this design space: $c \in \mathcal{DS}$ with $c_i \in M_i$ for $i \in I$.

The evaluation of a candidate for a extra-functional property q is a function $\Phi_q : \mathcal{DS} \to \mathcal{QA}_q$, where the codomain $\mathcal{QA}_q$ is the set of possible values of the extra-functional property q. Then, $\Phi_q(c)$ denotes the evaluated value of a extra-functional property q for a candidate $c \in \mathcal{DS}$. For example, when evaluating the mean response (mrt) time of a candidate, $\mathcal{QA}_{mrt} = \mathbb{R}_+$. When evaluating the probability of failure on demand (POFOD) of a candidate, $\mathcal{QA}_{pofod} = [0, 1]$. For example, for a specific candidate c, Φ_{pofod} could evaluate to $\Phi_{pofod}(c) = 0.005$

Not all candidates as defined above are valid for a system. We reflect this by including three types of *constraints*:

(i) **Contradiction Constraints**: Some design option values of different domains might be incompatible. For example, component B might not work with with one of the available application server implementations, e.g. with the Sun Glassfish implementation.

(ii) **Software Architect Constraints**: The software architect may have reasons to exclude some values of a design option or constrain a design option to a certain value. For example, he might chose to use the SOAP communication protocol for all communications in between the components, as this is a company-wide standard. Thus, all candidates using other protocols (e.g. RMI) are not considered in this case.

(iii) **Quality Requirements Constraints**: Requirements might be formulated that enforce a certain level for a extra-functional property, while other attributes are optimised. For example, the reliability in terms of probability of failure on demand of the system can be required to be lower than 0.05, while the target of the search is optimal performance.

The first two cases result in a number of candidates being invalid. As they refer to the design space, we call them *design space constraints*. We define a forbidden area on the design space DS:

$$DesignSpaceConstrained_{\mathcal{DS}} := \{c \in \mathcal{DS} \,| c \text{ is invalid}\}$$

As this set of tuples can also be seen as a mathematical relation, any language to specify relations could be used to allow an easier specification of the constraints than by enumerating all forbidden candidates. For some modelling languages and meta-(meta-)models, specialised constraint languages already exist. For example, if the candidate modelling uses the Meta-Object Facility [14] (MOF), we can use the Object Constraint Language [15] (OCL) for constraint specification. However, as the design space is finite, there is no difference in principle.

The third type of constraints applies to the codomains of the evaluation functions Φ_q. As they refer to the extra-functional properties of a candidate, we call them *extra-functional constraints*. To define the constraints, we denote $\leq_q$ as the total order on the extra-functional property domain $\mathcal{QA}_q$ for which

$$a \leq_q b \Leftrightarrow a \text{ is better than or equal to } b \text{ in terms of the extra} - \text{functional property } q$$

with $a, b \in \mathcal{QA}_q$

For example, a response time of 2 seconds is better than a response time of 5 seconds. For probability of expected service delivery on demand, 0.9 is better than 0.8. The order $>_q$ is defined as the opposite, but in this case strict order: $a >_q b \Leftrightarrow a$ is worse than b in terms of the extra-functional property q.

Note that there might be quality domains in which such an order does not come naturally, for example response time distribution functions. However, we can define functions on the extra-functional property domain and define the order based on that function. To continue the example, we could define a function of whether 90% of all requests are completed within 5 seconds on the response time distribution functions, and define the total order on the codomain {yes, no}.

Now, we can define extra-functional constraints. Let $req_q \in \mathcal{QA}_q$ be the requirement in terms of a maximum allowed value for extra-functional property q. Then, we define the set of all candidates $c \in \mathcal{DS}$ with an evaluated extra-functional property q worse than req_q:

$$ExtraFunctionallyConstrained_{\mathcal{DS}} := \{c \in \mathcal{DS} \,|\Phi_q(c) >_q req_q\}$$

Thus, the set of valid candidates is constrained by all these types of constraints:

$$Valid_{\mathcal{DS}} := \mathcal{DS} \setminus (DesignSpaceConstrained_{\mathcal{DS}} \cup ExtraFunctionallyConstrained_{\mathcal{DS}})$$

The optimisation problem for a single extra-functional property q then is to minimise $\Phi_q(c)$ while c is a valid candidate:

$$Opt_q : min_{\leq_q} \Phi_q(c) \text{ subject to } c \in Valid_{\mathcal{DS}}$$

For multi-criteria optimisation of different extra-functional properties we can either weight the single attributes creating a combined evaluation function Φ, or study Pareto optimality [6]. An interesting extra-functional property for multi-criteria consideration is the cost of a candidate. Here, trade-off decisions between performance and cost can be automatically explored, interesting candidates can be detected and presented to the software architect to make the final decision.

2.2 Automated Solution

The design-space consists of numerous, but countable number of candidates. The value domains of the design options are discrete. Thus, for the optimisation, we face a combinatorial optimisation problem [3, Def. 1.1]. Metaheuristics have been successfully applied to similar problems in software engineering [8].

Metaheuristics are general strategies that guide the search process using underlying problem-specific heuristics (see [3] for a more thorough characterisation). Examples are genetic algorithms, simulated annealing, or random-restart hill climbing [3]. The use of metaheuristics allows the automated search to pass over local optima that are not global optima. In contrast to that, pure rule-based approaches might be stuck in local optima and cannot make use of model parameters whose influence on the extra-functional property to be optimised is unknown.

Listing 1: Automated Extra-functional Property Improvement

```
Inputs:   $C \subset \mathcal{DS}$ A set of initial candidates
          $R = \{req_1, ..., req_n\}$ Requirements on further extra-functional properties
              $\{q_1, ..., q_m\}$
          $q_0$ extra-functional property that is to be optimised
          $P = \{\Phi_0, ...\Phi_m\}$ Evaluation functions for extra-functional properties
              $\{q_0, ..., q_m\}$

repeat {
    $C_{new}$ ←generateNewCandidates($C$, $DesignSpaceConstrained$, $R$, $P \setminus \Phi_0$);
    $C$ ← chooseNextIterationCandidates($C_{new}$, $C$, $\Phi_0$);
} until ( a stop criterion is fulfilled )
return $c$ ← bestOf($C$);
```

Listing 1 shows the general idea: metaheuristics iteratively create and evaluate new candidates. Thus, a current solution (or solution set) is evolved until some stop criterion is fulfilled (e.g. “no more improvements could be found in the last 10 steps”). During the search, additional constraints on the search space must be considered. Listing 1 illustrates the general idea based in terms of our design space problem for optimising a extra-functional property q_0 for a system and is explained below. As an aside, q_0 could also be a function that weights several extra-functional properties, e.g. "0.4 $*$ cost + 0.6 $*$ mean response time".

As long as better candidates are found with each iteration, the search advances. Each iteration consists of two steps: First, new candidates are created based on the current candidate(s) using generation *operations*. The candidate generation considers the additional constraints and produces only valid candidates. In the second step, promising candidates for future iterations are chosen based on the evaluation Φ_0 of q_0 that is to be optimised.

Many metaheuristics with different characteristics have been presented (see [3] for an overview). It still is an open question which metaheuristic configured with which parameters is the most appropriate for our problem.

For performance, there are design options with known influence on the overall system. For example, if we increase the processing speed of a highly utilised resource, the response time of a system will likely decrease (although there are exceptions). We formulate such performance domain knowledge into a set of heuristics. To generate new candidates, we can apply the resulting *heuristic operations*.

However, there are design options for which we have no prior knowledge on how they affect the extra-functional property at hand for a specific system. For example, if we replace a component A in a candidate system c' with a component B, we do not know in advance how the performance of the resulting candidate system c'_1 changes. Furthermore, the effects of a design option D_i for a candidate c are not necessarily independent of the other design option values $c_1, ..., c_{i-1}, c_{i+1}, ..., c_{|I|}$. If we consider a different candidate c'', that differs to c' in some design option values (e.g. $c'_1 \neq c''_1$ and $c'_2 \neq c''_2$, which may represent different allocation of the assembled components), exchanging A for B in c'' could have different effects (e.g. slowing down the system) than in c' (e.g. speeding up the system). Here, we want the metaheuristic to try solutions we have no prior knowledge of, thus we create *undirected operations* to generate new candidates without prior knowledge.

Undirected operations could include local search or random change. For local search, all neighbouring candidates (e.g. with one component exchanged, or with a processing resource speed increased by one increment) are evaluated. For random change, a number of design option values are changed for candidates to randomly chosen values.

3 Performance Improvement for Component-Based Systems

Although the concepts of automated performance improvement described in Section 2 are generic, our goal is to apply the approach on performance prediction for component-based software systems. This section explains how we have tailored the approach to deal with the specifics of component-based systems by applying it on the Palladio Component Model (PCM) (Section 3.1). We specifically describe the design options available for component-based systems (Section 3.2). Afterwards, we describe an initial, prototypical implementation of performance-oriented performance improvement for the PCM (Section 3.3).

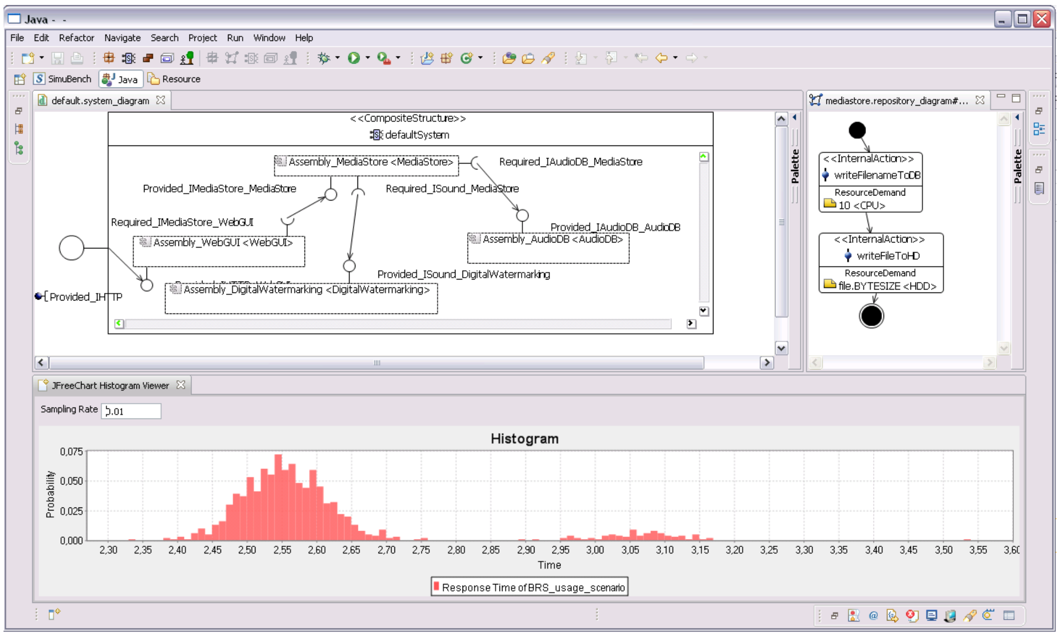

Fig. 1. A PCM Instance Example

3.1 Palladio Component Model

The PCM [2] is a modelling language specifically designed for performance prediction of component-based systems. The language allows the component developer specify the behaviour and the performance properties of components in a parametrised form. Then, software architects can combine these component specifications with specifications of component assembly, user behaviour, and resource environment to form an *architectural model*. A model transformation into a discrete-event simulation of generalised queueing networks allows software architects to predict various performance metrics such as utilisation or response time distributions of the system and of individual components.

The Eclipse-based PCM Bench tool allows the specification and analysis of PCM instances. Figure 1 shows an example PCM instance. In the figure, we see three diagrams (clockwise from upper left corner): the assembly of components to form a system, the activity-diagram-like performance specification of the component internals, and the predicted response time distribution.

3.2 Design Options for Component-based Systems

As components shall be black-box entities, which are possibly obtained from third parties, automated performance improvement may not change modelled internals of the components. Thus, we only vary component assembly, user behaviour, and resource environment. The following design options are currently present in a PCM instance:

- **Substituting Functionally Equivalent Components:** A component can be replaced by other available components with potentially better extra-functional properties. For that, a notion of substitutability needs to be present in the model. In the PCM, a component B can substitute a component A if all interfaces provided by A are also provided by B and B requires no more interfaces than A.

- **Component Allocation:** The allocation of components to hardware nodes is a design option that also has no functional impact, but may have a high extra-functional impact. For example, for performance, it is beneficial to distribute the load evenly on several servers or to allocate two components with tight communication together on one server to allow local communication.
- **Resource Environment:** We model the hardware environment separately from the component assembly to allow to adjust the sizing of resource. As the choice of hardware in the hardware environment has no functional impact, but a large non-functional impact on the system, we can change its configuration during the search. For example, the processing speed of a certain CPU can be increased.
- **Configuration Parameters:** The software itself can offer configuration parameters that have no functional influence, but may affect the performance of the system. Mainly, this applies for the middleware as offered by application servers. For example, the number of threads provided by an application server for the application can vary. Additionally, the communication between remote components (modelled as a connector) can be configured, for example the network protocol to be used can be changed (e.g. RMI or SOAP).
- **Usage Profile:** The number of users as well as their input parameters can have an effect on the overall performance of the system. These values are included within a PCM instance. Adding these values to the changeable parameters of the model, the scalability of the system at hand in relation to other design options (as presented above) can be studied.

3.3 Prototypical Implementation for the PCM

We have implemented the PerOpTeryx tool to provide an initial prototypical implementation of the ideas presented in section 2.2 for automated extra-functional property improvement in general and in section 3.2 for performance of component-based system. PerOpTeryx can automatically improve the performance of PCM instances. It is realised as a plugin for the Eclipse framework and thus seamlessly extends the PCM Bench. The PCM Bench relies on the Eclipse Modelling Framework (EMF) for modelling the component-based system. During the search, PerOpTeryx manipulates the EMF instance of the PCM to generate new candidates. To evaluate a candidate, its EMF instance is automatically transformed in a generalised queueing network system and analysed using a discrete-event simulation. Both transformation and analysis are provided by the existing PCM Bench implementation and described in detail in [2]. From the various resulting performance metrics, PerOpTeryx currently uses the mean response time of an entire usage scenario for assessing a candidate.

The current early version of PerOpTeryx applies steepest-ascent hill-climbing to the problem, which is a simple metaheuristic. For each iteration, the best candidate from the current set of solutions is chosen as the basis for new candidates. If no candidate from the current set of solution is better than their common direct ancestor, the search terminates. Thus, PerOpTeryx cannot pass over local minima. As we have no cost model integrated yet, the search also terminates if a candidate fulfils the specified performance requirements. In the prototype, we use predicted mean response time to state the performance requirements. To really find an supposedly optimal candidate, this requirement can be set to 0 seconds, then, only the first mentioned stop criterion will end the search.

For candidate generation, the tool includes one heuristic operation and one undirected operation as an example to illustrate the concepts. Technically, for each new candidate (for both operations), the EMF model of the current candidate is copied and then varied.

The *heuristic operation* "Increase processing speed" reflects our performance-domain knowledge that adding processing power improves performance. For example, it is possible to buy faster processors. However, this also introduces higher costs, thus we only want to increase the processing speed if needed.

The "Increase processing speed" heuristic states the following: Whenever a resource (e.g. a CPU) is found in the model that has a predicted utilisation of more than 75%, the processing speed of the resource is increased by 10%. This reduces the service time of all jobs executing on this resource.

The *undirected operation* "Replace components" has no conditions, as we do not know about the effects of replacing components in advance before trying them in the actual system. The component models are highly parametrised and components compete for resources. The rationale behind this operation is that multiple components for one task could be available from third parties and that some of them might lead to better performance.

PerOpteryx can replace components in the system if they are substitutable by other components in a component repository. For each component in the system, the prototype checks whether there are substitutions by iterating through all available components in the repository. Substitutability is checked based on the provided and required interfaces. As interfaces are first-class entities in the PCM, we can easily check whether component provide or require the same interface, i.e. refer to the same interface entity as "provided" or "required". For simplicity, we assume here that an interface fully defines the functionality, i.e. that components referring to the same interface entity as "provided" offer the same functionality through this interface. Then, component B can substitute component A in a system if and only if (1) B provides at least the interfaces A provides, (2) B requires at most the interfaces A requires, and (3) A is not identical to B. Technically, if a matching component is found, the prototype generates the new candidate by replacing the old component with the new component in the system model and by updating references to that component in the connectors. Thus, the new component is automatically allocated to the same server. Our current algorithm substitutes one component at a time (i.e. only considers neighbouring candidates and thus conducts a local search).

We aim to evolve the PerOpteryx into a framework for extra-functional property optimisation for component-based software systems.

4 Case Study

In this section, we present a case study applying PerOpteryx to the so-called Business Reporting System (BRS), which is based on an industrial system. With

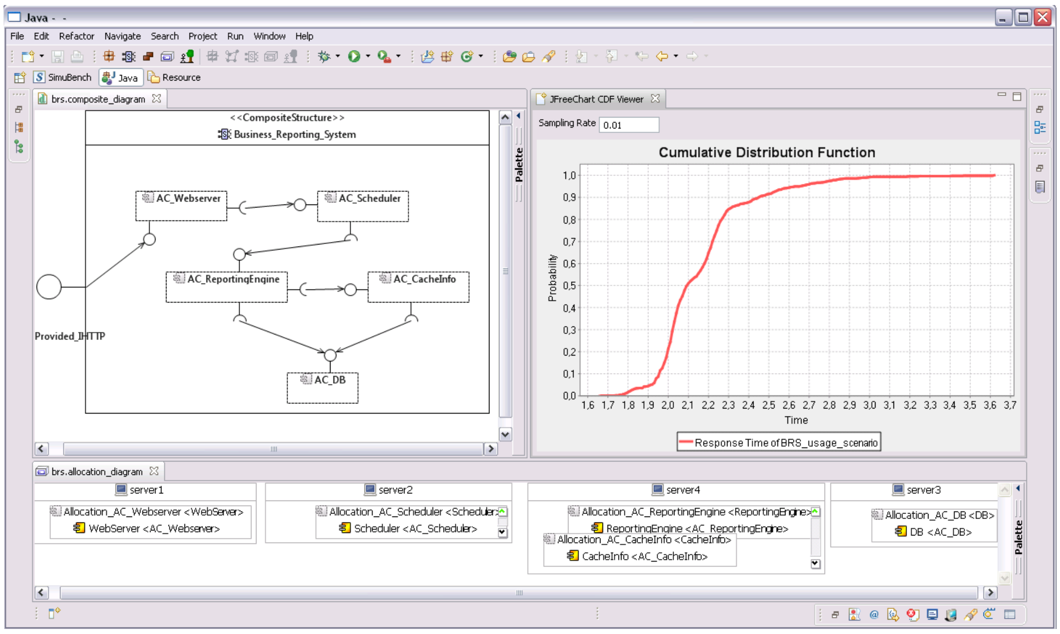

Fig. 2. The Business Reporting System modelled in the PCM

this case study, we demonstrate the feasibility of our approach and illustrate how the current prototype PerOpteryx works.

The BRS is a 4-tier, web-based system to monitor and manage business data. On a high abstraction level, it consists of 5 software components. Figure 2 shows how the BRS is modelled in the PCM[3]. Clients either request business reports or specific entries from the database via the `Webserver` component. A `Scheduler` component connects the `Webserver` component with the core application. The core application consists of a component `ReportingEngine`, which manages the creation of reports, and a component `CacheInfo`, which buffers data from the database for quick access. Both, the `ReportingEngine` and the `CacheInfo` query the component `Database`, which stores a configurable amount of entries in its tables. The `Webserver`, `Scheduler` and `Database` component are each allocated on a dedicated server (server 1 to 3), whereas the components `ReportingEngine` and `CacheInfo` are allocated together on server 4 (see allocation diagram in the lower part of figure 2).

We specified that the mean response time of the BRS for the expected usage should be less than 2.5 seconds. The usage was expected to be an open workload with a user arriving every 1.3 seconds. We specified no further constraints. For the `Webserver` component, there are two alternatives named `WebServer2` and `WebServer3` available in the repository.

After the system is modelled in the PCM, we can directly start the optimisation process. In the following, we will explain the steps PerOpteryx takes for our case study example. Figure 3 shows the candidates evaluated during the process.

[3] You can download the model at http://palladio-approach.net/_fesca2009

Candidate	mean response	Utilisation of			
	time (in sec)	server 1	server 2	server 3	server 4
BRS	3.43	0.85	0.19	0.18	0.78
BRS-Web3	2.52	0.14	0.19	0.18	0.79
BRS-Web3-incrCPU4	2.16	0.14	0.19	0.19	0.71

Table 1
Utilisation of the Servers for Selected Candidates

4.1 Initial analysis

We analyse the model of the initial candidate "BRS" with a discrete-event simulation developed for the PCM that can output the predicted mean response time, amongst other metrics. The analysis of the initial model results in a predicted mean response time of 3.43 seconds, which does not fulfil our performance requirement of 2.5 seconds. The utilisation of the servers is depicted in table 1, line "BRS".

4.2 First iteration of the search

(i) Generation of new candidates: First, the candidate is analysed for design options:
  - (a) *Replace components:* For each component in the system, the repository is searched for alternatives. In our case, two alternatives can be found for the `Webserver` component: `Webserver2` and `Webserver3`. Thus, two new candidates are created: In candidate "BRS-Web2", the `Webserver` component is replaced by the `Webserver2` component. In candidate "BRS-Web3", the `Webserver` component is replaced by the `Webserver3` component.
  - (b) *Increase processing speed:* The processors of both the web server (server 1) and the application server (server 4) are both rather highly utilised (see table 1), both reach the limit of 75% utilisation that is the current condition of our *Increase processing speed* heuristic. Thus, the processing rate of the highest utilised processor (server 1) is increased by 10% in candidate "BRS-incrCPU1".

(ii) Analysis of the new candidates: The simulations is conducted with the three new candidates. The results are shown in figure 3. The new candidate "BRS-Web3" exhibits a better mean response time than the initial candidate. Candidate "BRS-incrCPU1" let to no visible improvement. For candidate "BRS-Web2", the mean response time even worsened by factor 30: Here, the system is overloaded.

(iii) Selection of the best candidate: Candidate "BRS-Web3" has the lowest mean response time of 2.52 seconds, thus it is used as a basis for the next iteration.

(iv) Stop criteria: Candidate "BRS-Web3" narrowly misses the requirement of 2.5 seconds, thus the search continues.

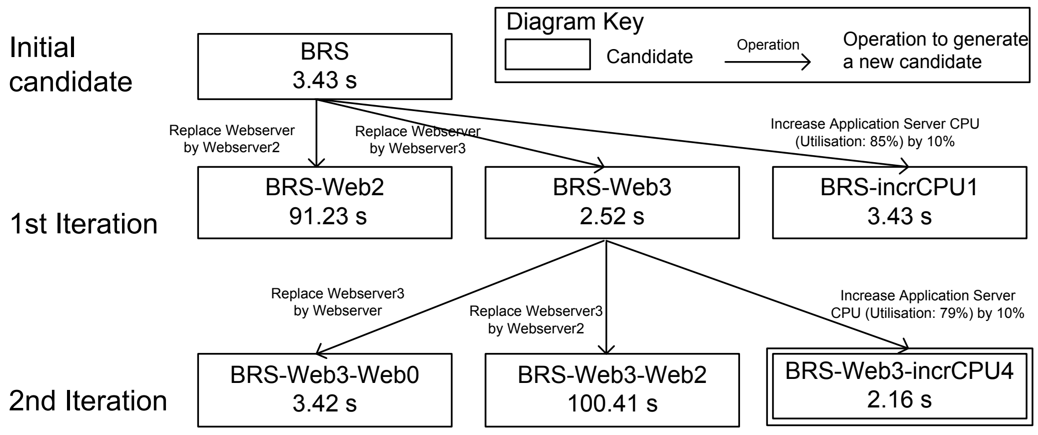
Diagram Key
Candidate
Operation
Operation to generate a new candidate
Initial candidate
BRS 3.43 s
Replace Webserver by Webserver2
Replace Webserver by Webserver3
Increase Application Server CPU (Utilisation: 85%) by 10%
1st Iteration
BRS-Web2 91.23 s
BRS-Web3 2.52 s
BRS-incrCPU1 3.43 s
Replace Webserver3 by Webserver
Replace Webserver3 by Webserver2
Increase Application Server CPU (Utilisation: 79%) by 10%
2nd Iteration
BRS-Web3-Web0 3.42 s
BRS-Web3-Web2 100.41 s
BRS-Web3-incrCPU4 2.16 s

Fig. 3. Result of the Performance Improvement

4.3 Second Iteration of the search

(i) Generation of new candidates. Based on candidate “BRS-Web3”, new candidates are generated.
   (a) *Replace components:* The possible alternatives are to change the `Webserver3` component back to `Webserver` or to switch to `Webserver2`. As both will result in the same candidate, we will not go into detail here.
   (b) *Increase processor speed:* The processor of server 1 is now much less utilised, however, the application server (server 4) processor still reaches the limit of 75% utilisation. Thus, the heuristic is applied creating candidate “BRS-Web3-incrCPU4”.

(ii) Analysis of the new candidates: The simulations is conducted for the new candidates. The results are shown in figure 3.

(iii) Selection of the best candidate: Candidate “BRS-Web3-incrCPU4” is an improvement compared to candidate “BRS-Web3”, thus it is chosen.

(iv) Stop criteria: Candidate “BRS-Web3-incrCPU4” fulfils the performance requirements of 2.5 seconds and the search terminates.

4.4 Results

The prototype successfully found a candidate that fulfils the response time requirements of 2.5 seconds. Figure 3 shows the 7 candidates analysed during the run. The automated performance improvement took up 6.3 minutes, whereas the simulation of each candidate needed between 21 and 38 seconds.

The tool outputs the means response time for all candidates and sorts them, fastest response time first. Additionally, the created candidates’ models are kept to allow the software architect further insight in what has been changed and also the individual simulation of each candidate.

5 Limitations and Future Work

In the following, we first discuss the current limitations of the prototype compared to the general idea of our approach and present ideas how to cope with them in future versions. Then, we give ideas for future extensions of the approach itself.

5.1 PerOpteryx Limitations

Local optima: The current prototype gets stuck in the closest local optima, even if it is not close to the global optimum. This limitation, however, can be easily met by applying more sophisticated metaheuristic such as simulated annealing, genetic algorithms or random-restart hill climbing.

Time consumption: The current performance evaluation of a single candidate using the discrete-event simulation is rather costly (ca. 30 seconds per candidate). As many candidates have to be analysed during a full-size optimisation run, the time needed for each single evaluation is critical. However, we think that an optimisation run of several hours would be acceptable for software architects, because it can save them hours or days of manual work. Additionally, several further options to speed up the analysis exist, for example, faster analysis techniques such as an existing LQN simulation can be used.

Design options: Not all design options for component-based systems (as presented in section 3.2) are supported by the current prototype. Adding the missing design options would further increase the number of possible candidates and thus increase the potential for design improvement.

Extra-functional properties: Only performance prediction is supported by the prototype so far. For architectural decisions, however, it is usually not appropriate to just optimise a single extra-functional property, because other extra-functional properties can worsen. Several more sophisticated metaheuristics exist that explicitly take several criteria into account and perform multicriteria Pareto optimisation, for example [7]. We plan to integrate such metaheuristics together with a simple cost model of initial cost and operating cost for both software and hardware as a next step.

Constraints: The specification of constraints and their consideration during the search is not yet supported. Here, OCL could be used to specify constraint on the underlying EMF model.

Performance metrics: So far, only the mean response time is supported. However, PCM predictions yield response time distributions for which requirements specified as quantiles are more useful (i.e. 90% of all requests must complete within 5 seconds) and should be supported.

5.2 Automated Performance Improvement Future Work

During the analysis of a system at hand, we might be able to learn correlations of design option values and performance for that specific system. For example, during the search, we can learn that component A is better than component B for most candidates. Possibly, we can even find further heuristics for undirected operations that are valid for many systems and create new heuristic operations. Some proposed metaheuristics such as [17] already incorporate such learning techniques that could be adopted for our automated performance improvement.

Another future extension for our approach is to add an interactive mode. The software architects could evaluate candidates during the search, so that their judgement would be a further criterion (interactive optimisation, [8]). Additionally, software architects could guide the search when several options how to evolve candidates are at hand and prioritise the available operations (comparable to interactive modes of model checker tools).

6 Related Work

Classical performance analysis tools for queueing network or stochastic Petri-nets usually only produce performance metrics after analysing the model and do not provide feedback on how to improve the model. The SPE-ED tools by Smith et al. [16], which relies on queuing network analysis, features a visualisation of the performance metrics in the graphical representation of a performance model by colouring performance-critical steps. While this provides a starting point for improving the model, there are no concrete guidelines to make these steps less performance-critical.

Xu et al. [18] present a semi-automated rule-based approach to find configuration and design improvement on the model level. Based on a Layered Queueing Network (LQN) model, performance problems in terms of bottlenecks and long paths are identified in a first step. Then, rules containing performance knowledge are applied to the detected problems. The search on the model level itself is fully automated, however, for some of the suggested design improvements, it is doubtful whether they actually can be implemented (e.g. the aforementioned improvement to somehow reduce the execution time for a task). The search stops when the performance requirements are met or if no more improvements can be found in all branches of the spanned tree of alternatives.

Cortellessa et al. [5] propose an approach for feedback generation for software performance analysis, which aims at systematically evaluating performance prediction results using step-wise refinement. The approach relies on the manual detection of performance anti-patterns in the performance model. There is no support to automatically solve a detected anti-pattern, and there is no suggestion of new architecture candidates.

Bondarev et al. [4] introduce the DeepCompass framework for design space exploration of embedded systems. The framework relies on the ROBOCOP component model. It uses a Pareto analysis to resolve the conflicting goals of optimal performance and low costs for different architecture candidates. Therefore, performance metrics for each architecture candidate are plotted against the costs of each candidate. The approach requires a manual specification of all architecture candidates and provides no support for suggesting new candidates.

McGregor et al. [13] have developed the ArchE framework. ArchE assists the software architect during the design to create architectures that meet quality requirements. It helps to create architectural models, collects requirements (in form of scenarios), collects the information needed to analyse the extra-functional properties for the requirements, provides the evaluation tools for modifiability or performance analysis, and suggests improvements. Compared to our work, ArchE only features a simple performance model and the architecture model is not component based.

Other than the former quantitative approaches, there are qualitative architectural evaluation methods, such as ATAM [10] or SAAM [11]. These approaches do not formally model software architectures, but instead rely on textual specification of usage scenarios. Therefore, these approaches only allow an informal discussion of performance properties and architecture candidates, but no automated support for exploring the design space.

Overall, the approach presented here is one example for search-based software engineering [8], where metaheuristic search techniques and optimisation are applied to problems in software engineering.

7 Conclusion

This paper presents a fully-automated approach to improve the expected performance of component-based software designs and a prototypical implementation for it. Using this approach, the design space spanned by different design options (e.g. available components and configuration options) can be systematically explored using metaheuristic search techniques and performance-domain heuristics. Based on an initial architectural model of a system, new candidates are automatically generated and evaluated for performance. We show a proof-of-concept case study to demonstrate the approach.

Using this approach, software architects can more easily create high-quality component-based software designs. The automation of the search saves the software architect effort for a manual exploration. Thus, the gap between applying formal performance prediction methods and actually improving the design of a system can be closed.

The first steps to extend our work will be the integration of a cost model and the use of multi-objective-enabled metaheuristics to support trade-off decisions among different extra-functional properties. The prototype will be extended with more design change operations such as changing the allocation of components and more performance domain heuristics. The integration of a framework for metaheuristic search will allow the comparison of several metaheuristics. In the long run, we aim to evolve the prototype into a framework for extra-functional property optimisation for component-based software systems.

References

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[3] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[4] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock. Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 153–163, New York, NY, USA, 2007. ACM Press.

[5] V. Cortellessa and L. Frittella. A framework for automated generation of architectural feedback from software performance analysis. In K. Wolter, editor, *Formal Methods and Stochastic Models for Performance Evaluation, Fourth European Performance Engineering Workshop, EPEW 2007, Berlin, Germany, September 27-28, 2007, Proceedings*, volume 4748 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2007.

[6] M. Ehrgott. *Multicriteria Optimization.* Springer-Verlag, New York, USA, 2005.

[7] H. Eskandari and C. D. Geiger. A fast pareto genetic algorithm approach for solving expensive multiobjective optimization problems. *Journal of Heuristics*, 14(3):203–241, June 2008.

[8] M. Harman. The Current State and Future of Search Based Software Engineering. *Future of Software Engineering, 2007. FOSE '07*, pages 342–357, May 23-25 2007.

[9] J. R. Jackson. Networks of waiting lines. *Operations Research*, 5(4):518–521, 1957.

[10] R. Kazman and L. Bass. Categorizing business goals for software architectures. Cmu/sei-2005-tr-021, Carnegie Mellon University, Software Engineering Institute, Dec. 2005.

[11] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A method for analyzing the properties of software architectures. In B. Fadini, editor, *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[12] H. Koziolek and V. Firus. Empirical Evaluation of Model-based Performance Predictions Methods in Software Development. In R. H. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, editors, *Proc. 1st Int. Conf. on the Quality of Software Architectures (QoSA'05)*, volume 3712 of *Lecture Notes in Computer Science*, pages 188–202. Springer-Verlag Berlin Heidelberg, 2005.

[13] J. D. McGregor, F. Bachmann, L. Bass, P. Bianco, and M. Klein. Using arche in the classroom: One experience. Technical Report CMU/SEI-2007-TN-001, Software Engineering Institute, Carnegie Mellon University, 2007.

[14] Object Management Group (OMG). MOF 2.0 Core Specification (formal/2006-01-01), 2006.

[15] Object Management Group (OMG). Object Constraint Language, v2.0 (formal/06-05-01), 2006.

[16] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2002.

[17] C. H. M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberger, and H.-P. Schwefel, editors, *PPSN*, volume 1141 of *Lecture Notes in Computer Science*, pages 494–503. Springer, 1996.

[18] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2008. ACM.