PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization

Anne Koziolek*, Heiko Koziolek†, Ralf Reussner*
*Karlsruhe Institute of Technology, Karlsruhe, Germany
Email: {koziolek,reussner}@kit.edu
†ABB Corporate Research, Ladenburg, Germany
Email: heiko.koziolek@de.abb.com

ABSTRACT

Designing software architectures that exhibit a good trade-off between multiple quality attributes is hard. Even with a given functional design, many degrees of freedom in the software architecture (e.g. component deployment or server configuration) span a large design space. In current practice, software architects try to find good solutions manually, which is time-consuming, can be error-prone and can lead to suboptimal designs. We propose an automated approach guided by architectural tactics to search the design space for good solutions. Our approach applies multi-objective evolutionary optimization to software architectures modelled with the Palladio Component Model. Software architects can then make well-informed trade-off decisions and choose the best architecture for their situation. To validate our approach, we applied it to the architecture models of two systems, a business reporting system and an industrial control system from ABB. The approach was able to find meaningful trade-offs leading to significant performance improvements or costs savings. The novel use of tactics decreased the time needed to find good solutions by up to 80% .

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*modelling techniques*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design, Architecture, Optimisation, Performance

1. INTRODUCTION

Software architecture design is influenced to a large extent by the consideration of quality attributes, such as performance, reliability, maintainability, costs, or security. Experienced software architects intuitively know styles and tactics to improve quality attributes of a software architecture [3]. In recent years, many researchers have proposed to encode architectural design decisions into software architecture models (e.g., using architecture description languages or UML) [28] thus enabling automated reasoning. A number of approaches evaluate these models for performance [2, 17] in terms of expected response times, throughputs and resource utilizations. This systematic support can lead to better decisions than experience [21].

As a major challenge in this area, most evaluation tools are only able to determine the quality attribute values (e.g. 5 sec response time) for a given architectural model. They leave the task of improving the architectural model as a manual exercise to the software architect. Due to the large design space for non-trivial systems and many degrees of freedom, improving the architecture is an error-prone and tedious task [3]. Isolated improvement of a single quality attribute can result in degradation of other quality attributes, which is hard to determine and quantify by software architects manually.

While a completely synthesised design is infeasible, many degrees of freedom that influence quality attributes remain in the software architecture even after functional design. For example, the component deployment, hardware sizing, component selection, and possibly further configuration options of components, servers, and middleware can be adjusted. To automate the improvement of architectural models within such degrees of freedom, researchers have proposed rule-based and metaheuristic approaches. Rule-based approaches [30, 10] translate known tactics, such as bottleneck removal or caching, into processable rules for manipulating architectural models. However, these rules are restricted to one quality attribute, without taking the degradation of others into account. Metaheuristic approaches [1, 5, 23] encode the architecture model improvement as an optimization problem and apply general-purpose, problem-independent optimisation strategies such as evolutionary algorithms, hill climbing, or simulated annealing [8]. Existing approaches in this direction operate on an encoding of the problem that does not contain any additional domain-specific knowledge.

We present a novel hybrid approach called PerOpteryx that incorporates architectural performance tactics into a metaheuristic optimization process. To the best of our knowledge, it is the first such approach for software architecture design optimization. PerOpteryx manipulates architec-

QoSA+ISARCS'11, June 20–24, 2011, Boulder, Colorado, USA.

ture models specified with the Palladio Component Model (PCM) [4] and uses the multi-objective evolutionary algorithm NSGA-II [9] internally. For performance analyses, it uses expressive Layered Queueing Networks (LQN) [11]. Other quality analyses (e.g., reliability analysis) are foreseen in PerOpteryx [20] and the tool can be flexibly extended with additional analysis or simulation tools.

With our approach, software architects do not have to search for alternative solutions manually. Instead, they can focus on the automatically determined optimal trade-offs between the considered quality attributes and choose the best trade-off for their situation. As the approach works on the architectural model level (as opposed to the e.g. performance model), architects can directly understand and use the automatically found solutions.

To validate our approach we have applied PerOpteryx in two case studies: a distributed business reporting system (BRS) and an industrial control system from ABB. Both cases show that PerOpteryx can find a meaningful set of optimal trade-off solutions. Additionally, we have compared the formerly unguided search with the tactics-guided search and found that the incorporation of a limited number of tactics already yielded an average decrease of optimization runtime by 80 percent (BRS) and 56 percent (ABB).

The contribution of this paper is a hybrid rule-based and metaheuristic optimization approach for software architecture models. We have implemented the automated application of a number of known architectural performance and costs tactics in our PerOpteryx tool and applied it on two case studies. Our approach improves the state of the art beyond former approaches because (i) it exploits multiple degrees of freedom (e.g. component allocation, component selection, hardware sizing) to find better solutions and can be extended to cover more, (ii) it improves multiple quality attributes at once and (iii) it incorporates established architectural tactics (e.g. "spread the load", "scale out bottleneck resources", etc.) to search the design space more efficiently.

This paper is organised as follows: Sec. 2 provides a general overview of our PerOpteryx approach. Sec. 3 presents a list of generic performance and costs architectural tactics and shows how they can be integrated into the PerOpteryx process. Sec. 4 validates the improvements of our approach in two case studies before Sec. 5 discusses related approaches for software architecture model improvement. Sec. 6 concludes the paper.

2. PEROPTERYX FRAMEWORK

This section introduces the models used in our approach and the steps of the automated design improvement process. We implemented PerOpteryx based on the Palladio Component Model (PCM) [4], but its concepts of searching the design space and applying tactics are not tied to the PCM and could also be applied on other modelling languages (e.g., UML MARTE [25]).

2.1 Running Example

For a quick overview of the PCM, consider the running example shown in Fig. 1 using a UML visualization. Software architects can model their systems as an assembly of software components deployed on hardware resources. The example model contains three software components deployed on two servers. For each service provided by a software component, an abstract behaviour model called *resource demanding service effect specification* (RDSEFF) is provided. It models demands to hardware resources (e.g., 2 seconds), and architecture-level control flow with branch probabilities p. The lower part of Fig. 1 shows one such model for each component. Details can be found in [4].

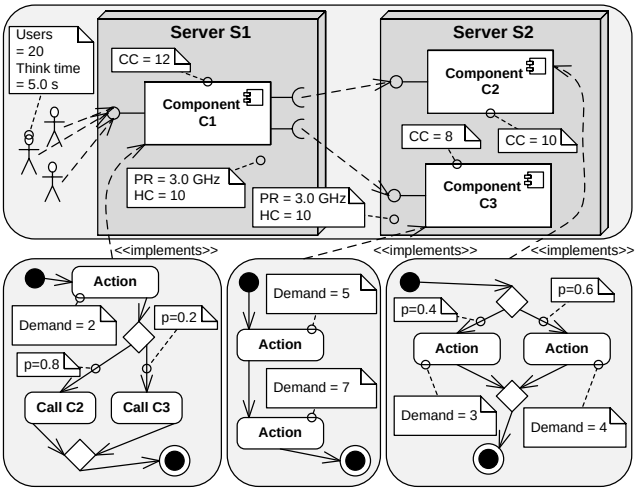

Figure 1: Running Example

Architects can annotate the hardware resources in PCM models with processing rates (PR) and hardware costs (HC) (cf. Fig.1). Furthermore, software components can be annotated with component costs (CC). Based on this information, different tools analyse the overall performance (i.e., response time, throughputs, utilizations) and costs (i.e., the sum of all component and hardware costs) of a PCM model.

RDSEFFs from the PCM can be explicitly parametrized for the influence factors to performance, e.g. resource environment, usage, and assembly of components [4, 19]. This means that the performance models remain valid if these influence factors are changed (e.g. faster servers, different workload, altered component topology).

In our case, the performance analysis for PCM models is based on the PCM2LQN [18] transformation into layered queueing networks (LQN) [11], which is a popular and expressive performance model. The costs analysis is based on the PCM2Costs transformation [20]. For the running example, the tools calculate an expected mean user response time of 8.8 seconds, a utilization U(S1) of 17% for server S1, a utilization U(S2) of 88% for server S2 and costs of 407 monetary units.

2.2 PerOpteryx Search Process

The PerOpteryx approach applies an automated, metaheuristic search process on a given PCM model to improve performance p and costs c properties. Because we consider multiple, possibly conflicting quality attributes, we search for Pareto-optimal candidates [8]: A candidate is Pareto optimal iff there exists no other candidate that is better in all quality criteria. The result of the optimisation is a Pareto front: A set of candidates that are Pareto optimal with respect to other candidates evaluated so far, and which should approximate the set of globally Pareto-optimal candidates well.

The process steps of PerOpteryx are depicted in Fig. 2.

In **step 1** the degree of freedom instances in the model are automatically identified based on rules that describe degree

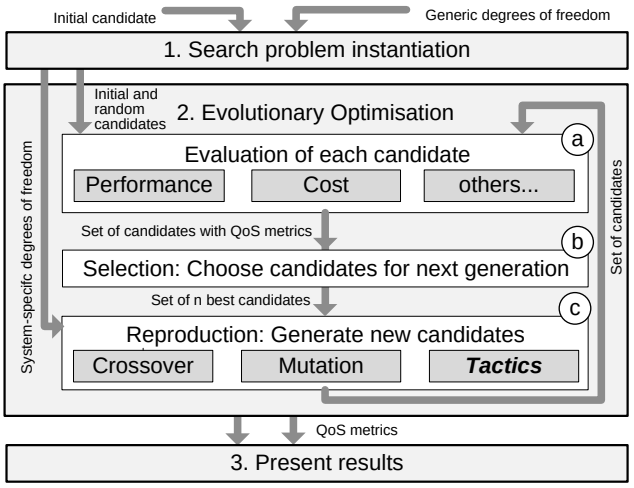

Figure 2: PerOpteryx process model

of freedom types (e.g. the degree of freedom type "component allocation") for the PCM. In our running example, there are five degree of freedom instances: changing the processing rates (for 2 servers), and changing the component allocation (for 3 components). In more complex PCM models, the degrees of freedom may for example also comprise the selection of different components, the configuration of middleware parameters, or the adjustment of thread pool sizes. The degree of freedom instances define the search space for PerOpteryx. Each degree of freedom has a set of design options. For example, let us assume server S1's speed can be varied between 2GHz and 4GHz. Each possible solution can be represented as a set of decisions, one for each degree, called genome. For example, the initial PCM model has a genome of (PR of S1: 3GHz, PR of S2: 3GHz, allocation of C1: S1, allocation of C2: S2, allocation of C3: S2) for this search space. The resulting optimisation problem (with min denoting Pareto optimality and x a genome) for this example is:

$$Opt : \min(p(x), c(x)) \text{ with } x \in [2GHz, 4GHz]^2 \times \{S1, S2\}^3$$

In general, for a set of degree of freedom instances D where each $d \in D$ has a set of design options O_d, and for a set of quality properties Q to be minimised that are evaluated by a combined evaluation function $\Phi : \Pi_{d \in D} O_d \rightarrow \mathbb{R}^{|Q|}$, the problem is

$$Opt : \min \Phi(x) \text{ with } x \in \Pi_{d \in D} O_d$$

In **step 2** PerOpteryx applies evolutionary optimization based on the genomes. The evolutionary optimization is based on the NSGA-II algorithm [9], which is one of the advanced elitist multi-objective evolutionary algorithms, and suitable for our multi-objective combinatorial problems [8]. In addition to the initial PCM model genome, PerOpteryx generates several candidate genomes randomly based on the degree of freedom instances as the starting population. Then, iteratively, the main steps of evaluation (step 2a), selection (step 2b), and reproduction (step 2c) are applied.

First, each candidate is evaluated by generating the PCM model from the genome and then applying the LQN and costs solvers (**step 2a**) on it. The LQN solver applies an efficient, heuristic algorithm on the queueing model and deduces for example the expected response times for the

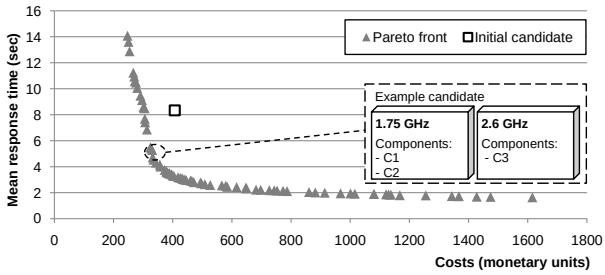

Figure 3: Pareto front for the simple example: Performance vs. costs

system. Other performance analysis approaches could be plugged in here, too.

Based on the evaluation results, the most promising candidates (close to the current Pareto front and well spread) are selected for further manipulation, while the least promising candidates are discarded (**step 2b**). During reproduction (**step 2c**), PerOpteryx manipulates the selected candidate genomes using crossover and mutation, and creates a number of new candidates. With cross-over, the genotypes of two selected candidate solutions are merged into one, for example by combining the processing rates from one candidate with the allocation by another candidate. With mutation, PerOpteryx varies one or more design options. For example, PerOpteryx might change the component deployment to allocate all components on one server, or increase the processing rate of server S2.

In **step 3** PerOpteryx presents the results of the evolutionary optimization to the software architect. Applying PerOpteryx on the running example yields 77 Pareto-optimal candidates (also called Pareto front) after 140 iterations. Fig. 3 visualises the front for performance and costs. The software architect can identify interesting solutions in the Pareto front fulfilling the user requirements and make well-informed trade-off decisions. To support this task, researchers have developed an number of multi-criteria decision analysis methods, such as multi-attribute utility theory, analytic hierarchy process, weighting methods, outranking methods and fuzzy methods [16], which can be incorporated into the PerOpteryx framework.

Architectural tactics could have sped up the search process for the running example. The search process could have started to improve the component deployment on the bottleneck resources (server S2 with $U = 0.88$), as it is generally known that removing bottlenecks improves performance [27]. Therefore, we describe the integration of performance and costs tactics in the PerOpteryx framework in the next section.

3. INTEGRATING TACTICS

Architectural tactics for quality attribute improvement of software architectures encode design knowledge and rules of thumb [3]. They are intuitively applied by experienced architects when designing an architecture.

We consider tactics on the level of the *software architecture* at design time, particularly in the domain of component-based distributed systems. As our approach targets improving an architectural model instead of an implementation, we exclude code-level tactics here. We may apply rules only on a PCM model, which describes a system

as an assembly of component and connectors, component behaviour, and component deployment to hardware nodes and reflects the available knowledge of the system at design time (Section 2.1, Details in [4]).

The following subsections provide a list of generic tactics for performance (Sec. 3.1) and costs (Sec. 3.2) and a sketch how these tactics can be mapped to PCM models. Finally, Sec. 3.3 describes in detail how the tactics are integrated in the PerOpteryx process.

3.1 Performance Tactics

We have aggregated our list of performance tactics in Table 1 from multiple sources about performance improvement on the level of architecture models with performance annotations. Smith and Williams [27] highlight technology-independent performance principles, patterns and anti-patterns. Further rules have been integrated from Microsoft's performance improvement guide [24] and literature on architectural tactics [3, 28]. The list tries to be comprehensive, but does not claim completeness. The tactics are grouped into software, hardware, and network tactics. The third column in describes how the rules can be applied to PCM models.

Classical performance analysis guides [14, 22] focus on queueing models and simulation, but provide only limited hints on how to improve performance on an architectural level. Contrary to other approaches (e.g., [30, 26]) our list of performance tactics is not tied to a specific performance model, such as LQN, or to a specific technology, such as EJB, but is more generically applicable.

Our approach assumes a *component-based* development process, where possibly black-box components from third party vendors are assembled. In such a process, it might be complicated to change the implementation of individual components as the code may not be accessible. Therefore, we have marked tactics that require to alter component implementations as "*Change component*" in the table. These tactics may therefore not be applicable in all cases.

The short rule descriptions in column three of Table 1 can be implemented to directly manipulate PCM models. Despite their brevity, some of the rules encapsulate complex relationships. For example, different kinds of database performance improvements, such as query optimizations or different schema layouts, are summed up in the tactic "Data structure and Algorithms", because in an architectural model such as the PCM, these changes are reflected only in changes to the resource demands of services of a database component. The large number of known concurrency patterns is summed up in the tactic "Concurrency".

As a proof of concept we have implemented the following performance tactics from Table 1 in our PerOpteryx tool, because they are most generic and applicable. For each tactic we detail rationale, precondition, action, additional effects, and available extensions below. Additionally, we illustrate each tactic based on the running example (from Fig. 1).

- **Component reallocation:** In distributed systems, components can be allocated to different servers. To improve performance in situations with high load, the overall load should be spread evenly across the system. Thus, some components should be reallocated from highly utilised servers to servers with low utilisation. If the right components are reallocated, this tactic can improve performance, while being cost-neutral.

	Name	Rule [Principle from [37]]	Modelling in Palladio CM
Software	Asynchronous Communication	Let components exchange data asynchronously to avoid synchronization delays. ["Parallel Processing Principle"]	*Change components:* change interfaces and RDSEFFs of blocked components to support asynch. comm., add cost.
	Caching	Keep the most frequently used data in a cache in main memory to allow quick access. ["Centering Principle"]	Create a cache component either immediately serving a request with a cache hit probability or delegating the request, add costs.
	Concurrency / Parallelisation	Introduce parallelism using multithreading or multiple processes. ["Parallel Processing Principle"]	*Change components:* use fork actions in RDSEFFs and reduce resource demand per thread, add costs.
	Coupling and Cohesion	Ensure a loosely coupled design that exhibits an appropriate degree of cohesion. ["Locality Principle"]	*Change components:* Merge components with a high interaction rate. Build subsystems, add costs.
	Internal Data Structures and Algorithms	Use appropriate data structures and algorithms within the components. ["Centering Principle"]	Identify component with highest resource demand and exchange them with different component implementations.
	Fast Pathing	Find long processing paths and reduce the number of processing steps. ["Centering Principle"]	Introduce additional components to serve the most frequently used functionality in a dedicated way, add costs.
	Locking Granularity	Acquire passive resources late and release early, minimize locking. ["Shared Resources Principle"]	*Change components:* change RDSEFFs and minimize the time between Acquire and Release Actions, add costs.
	Priorisation	Partition the workload and prioritize the partitions so that they can be efficiently queued. ["Centering Principle"]	Not yet supported.
	Resource Pooling	Ensure effective use of pooling mechanisms (Objects, Threads, Database connections, etc.). ["Fixing-Point Principle"]	Identify passive resources with the highest waiting delay and adjust their capacity.
	State Management	Use stateless components where possible to keep them decoupled and allow scalability. ["Shared Resources Principle"]	Not yet supported.
Hardware	Component Reallocation	Allocate software components from saturated resources to underutilized resources. ["Centering Principle"]	Identify resources with U>=maxThreshold & reallocate components to resources with U<=minThreshold
	Component Replication	Start multiple instances of the same component and spread the load on multiple servers. ["Spread-the-load Principle"]	Identify components accessed by many users, create multiple component instances and introduce load balancer component.
	Faster Hardware	Buy faster hardware to decrease the node utilization and response times. ["Centering Principle"]	Increase processing rate of bottleneck processing resources, increase hardware costs
	More Hardware	Buy additional servers and spread the load among them. ["Spread-the-load Principle"]	Increase the number of processing resources, introduce load balancer (incl. costs), increase hardware costs
Network	Batching	Avoid network accesses by bundling remote requests. ["Processing vs. Frequency Principle"]	Insert messaging components that bundle remote requests to batches and unpack them at the receiver side, add costs.
	Localization	Allocate frequently interacting components on the same hardware devices. ["Locality Principle"]	Identify components with a high interaction rate and reallocate them to the same resources.
	Remote Data Exchange Streamlining	Decrease the amount of data to be send across networks (e.g., using compression). ["Centering Principle"]	Create a compression component that shrinks the size of the data transferred, but adds a resource demand to the CPU.

Table 1: Performance Tactics

Precondition: The utilisation difference between the most saturated server S_b and the server with the lowest utilisation S_l is larger than threshold Δ_{realloc}: $U(S_b) - U(S_l) > \Delta_{\text{realloc}}$) and S_b hosts several components.
Action: One of the components allocated to server S_b is randomly chosen and reallocated to server S_l.
Additional effects: The reallocation is cost-neutral. However, it may introduce additional network processing overheads if components are separated that communicate intensely.
Example: In the running example, component C3 could be reallocated from server S2 (utilisation 88%) to server S1 (utilisation 17%).
Extensions: The component to reallocate could be chosen more intelligently by considering the demand of a compo-

nent and its communication frequencies with other (local and remote) components.

- **Faster hardware:** Highly utilised bottleneck servers slow down the system and should be scaled up. This tactic is limited by the maximally available resource speed.
  Precondition: The most saturated server S_b is utilised above a threshold ($U(S_b) >= U_{\text{faster}}$).
  Action: Increase the processing rate of server S_b by an increase factor f. If the result is higher than the maximum processing rate, choose that maximum. If the processors are chosen from a discrete set, choose the cheapest processor with a processing rate $PR > PR(S_b) \cdot f$.
  Additional effects: Hardware costs are increased. The modelled costs need to reflect additional drawbacks of the faster hardware such as e.g. increased maintenance costs.
  Example: The processing rate of the bottleneck server S2 could be increased by $f = 25\%$.
- **More hardware:** Due to the hard limit of server processing rates it is necessary to add additional servers (scale out) to cope with high loads. However, scaling out is limited by the software design. Currently, we consider the maximum number of servers to be the number of components (i.e. in the maximum scale-out each component is deployed to one dedicated server). This tactic is not effective if a single component causes most of the load in the system.
  Precondition: The most saturated server S_b is utilised above a threshold ($U(S_b) >= U_{\text{more}}$) and the maximum number of servers has not yet been reached.
  Action: Reallocate one component from the bottleneck server S_b to a new server.
  Additional effects: Increases hardware costs and possibly adds a performance overhead for the additional network communication.
  Example: A third server S3 could be added and component C3 could be reallocated to it.
  Extension: The extensions of the reallocation tactic also apply here. Additionally, a single component could also be deployed to multiple servers using load balancing techniques and possibly synchronisation strategies (both not yet supported by the PCM).

PCM models can already be improved for performance with these three tactics, as demonstrated in Sec. 4. We intend to extend our approach to comprise more tactics in future work.

3.2 Cost Tactics

Minimisation of costs is of high business interest. Here, we only consider costs that can be predicted based on the software architecture model as presented in Sec. 2: These can be development costs, procurement costs and operating costs, for example. Costs can be minimised by choosing a less expensive option for a degree of freedom. Examples are the choices for cheaper components or cheaper hardware. Additionally, all tactics that improve performance and increase costs can be inverted. In this work, we consider two costs tactics of this type:

- **Slower Hardware:** Inversely to the "Faster Hardware" tactic, this tactics decreases resource speeds of infrequently used servers, because we expect that performance is only slightly degraded, while costs are saved. This tactic is only applicable if faster servers are also more expensive.

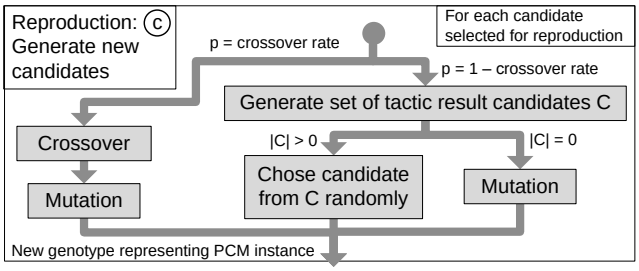

Figure 4: Integration of heuristics into the reproduction step. Cf. Fig 2 for an overview of the complete process

  Precondition: The server with the lowest utilisation (S_l) is utilised less that a threshold ($U(S_l) >= U_{\text{slower}}$).
  Action: Decrease the processing rate PR of S_l by an decrease factor d. If the result is lower than the minimum processing rate of the resource, choose that minimum. If the processors are chosen from a discrete set, choose the fastest processor S' with $PR(S') < PR(S_l) \cdot d$.
  Additional effects: Performance is degraded.
  Example: The processing rate of S1 ($U(\text{S1}) = 17\%$) could be decreased by $d = 25\%$ in the example.
- **Consolidate Servers:** Inversely to the "More hardware" tactic, lowly utilised servers can also be consolidated and their components can be joined one server to save cost.
  Precondition: The utilisation of two servers S_{l1} and S_{l2} is lower than a threshold ($U(S_{l1}) < U_{\text{cons}}$ and $U(S_{l2}) < U_{\text{cons}}$).
  Action: Reallocate all components from S_{l2} to S_{l1}, so that S_{l2} is no longer used.
  Additional effects: Performance may deteriorate. Also see reallocation tactic in Sec. 3.1.
  Example: Assume that the load of the running example was lower and both servers had a utilisation of lower that 25%. Then, all three components could be allocated to server S1, and the costs of S2 could be saved.

3.3 Tactics as Heuristic Operators

The tactics act as heuristic operators in the reproduction step (**step 2c**) of the PerOpteryx approach (Fig. 4). The evolutionary algorithm can perform a crossover, a mutation, or apply tactics on the selected candidates (parents) in order to improve the quality of the population. In PerOpteryx, the probability of a crossover is determined by a user-configurable crossover rate. PerOpteryx applies tactics if the algorithm chooses not to perform a crossover. Then, if the precondition of a tactic is fulfilled by a parent, a new candidate is generated based on the tactics by changing the parent's genome accordingly, and added to the set of tactic result candidates C. If no tactic precondition matches, set C remains empty and PerOpteryx performs a mutation of the parent's genome.

If the preconditions of multiple tactics match, PerOpteryx generates multiple candidates. To decide for one candidate, we assign weights between 0 and 1 to both the tactics (weights W) and the candidate (weights V).

Tactic weights W_t are assigned to each tactic t and define how promising it is in general. Candidate weights V_{c_t} are assigned to a generated candidate c_t based on the input candidate's applicability for tactic t. Then, PerOpteryx chooses

one candidate from candidate set C. Each candidate c_{t*} is chosen with probability:

$$Prob(c_{t*}) = \frac{W_{t*} \cdot V_{c_{t*}}}{\sum_{c_t \in C} W_t \cdot V_{c_t}}$$

Based on our experiments, we chose the following candidate weights V_{c_t} for our current tactics. Let S_b be the server with the highest utilisation, S_{l1} be the server with the lowest utilisation, and S_{l2} be the server with the second lowest utilisation. $U(S)$ denotes a server S's utilisation.

- **Component reallocation:** $V_{\text{realloc}} = U(S_b) - U(S_l)$. In our running example, we get a weight of $0.88 - 0.17 = 0.69$ for reallocating C3 to S1.
- **Faster hardware:** $V_{\text{faster}} = \frac{U(S_b) - U_{\text{faster}}}{1 - U_{\text{faster}}}$. In our running example, if U_{faster} is 80% we get a weight of $\frac{0.88-0.8}{1-0.8} = 0.4$ for the heuristic candidate with a higher processing rate of server S2.
- **More hardware:** $V_{\text{more}} = \frac{U(S_b) - U_{\text{more}}}{1 - U_{\text{more}}}$. In our example, if U_{more} is 80% we get a weight of $\frac{0.88-0.8}{1-0.8} = 0.4$ for adding a third server.
- **Slower Hardware:** $V_{\text{slower}} = \frac{U_{\text{slower}} - U(S_{l1})}{U_{\text{slower}}}$. In our example, if U_{slower} was 25%, we get a weight of $\frac{0.25-0.17}{0.25} = 0.32$ for decreasing S1's processing rate.
- **Consolidate Servers:** $V_{\text{cons}} = \frac{U(S_{l1}) + U(S_{l2})}{2U_{\text{cons}}}$.

This approach allows us to take both the expected impact of a tactic and its applicability to a concrete input candidate into account. Directing the evolutionary search process using tactics can speed up the search compared to an unguided search and find better candidates faster as will be evaluated in the following section.

4. EVALUATION

This section validates the benefits of our approach. Section 4.1 presents the goals of the validation, before Section 4.2 describes the setup of the optimization runs. We have applied the approach in two case studies, a business reporting system (BRS) (Section 4.3) and an industrial control system (ICS) from ABB (Section 4.4), which shows the industrial applicability of our appproach. Finally, Section 4.5 discusses the results and findings of the case studies.

4.1 Validation Goal

The goal of our evaluation is (i) to validate the applicability of our approach, i.e. that PerOpteryx is able to find a meaningful Pareto front, and (ii) to show that tactic-guided optimization runs are significantly superior to unguided optimization in terms of quality and efficiency.

To operationalize the second goal, we defined a coverage metric C to determine the quality of the found candidates and a speed-up metric D to determine the efficiency of the optimization runs. The coverage metric C (similar to the coverage indicator described in [31]) compares two Pareto fronts A and B (e.g. generated by a guided and an unguided search). First, the Pareto front P of $A \cup B$ is calculated. Then the coverage $C(A, B)$ of A is defined as the share of candidates from A in the P: $\frac{|A \cap P|}{|P|}$. Therefore if $C(A, B) > 0.5$ then A is a better Pareto front because of a higher number of candidates in P.

The speed-up metric D determines how many iteration steps earlier one optimization run has found a solution with equivalent quality. Because each iteration has a similar duration, this measures the computational effort of a run while it is independent of execution time measurement errors such as additional load on the executing machine. At each iteration i, an optimization run R has a current Pareto front of architectural candidates $P(R, i)$. To compare a tactics-guided run T with an unguided run B, we determine the smallest iteration step x in which the guided run T has a Pareto front $P(T, x)$ that is superior or equivalent to the unguided run B at the final iteration i_{max} (front $P(B, i_{max})$): $C(P(T, x), P(B, i_{max})) > 0.5$. For a fair comparison, we also determine the smallest iteration y in which the unguided run has already found a front $P(B, y)$ that is equivalent to the front $P(B, i_{max})$: $C(P(B, y), P(B, i_{max})) \geq 0.5$. Then, the guided run has found an equivalent solution $y - x$ iterations earlier. D is defined as the relative runtime improvement $\frac{(y-x)*100}{y}$ ([1]).

To assess the statistical significance of our results, we analysed the results for both coverage metric C and speed-up metric D using Student's t-test as implemented in the `t.test` procedure of the tool R for statistical computing (www.r-project.org).

4.2 Setup of the Optimization Runs

To account for the stochastic nature of evolutionary algorithms, we analysed 10 tactics-guided optimisation runs T_r, $0 < r < 9$, each starting with the initial candidate and 19 random candidates (different for each run) as population p_r. PerOpteryx was configured with $i_{max} = 200$ iterations, as initial experiments showed that the Pareto fronts do not change much afterwards, population size 20, and crossover rate 0.75. Then, each optimization run evaluated around 2000 candidates and ran for 5 to 6 hours on one 2.4 GHz core of a standard PC.

To compare the quality and duration of tactic-guided optimization (T) with unguided optimization (B), we ran another 10 unguided optimisation runs B_r, $0 < r < 9$, each starting with the same population p_r as its guided counterpart T_r. Then, we can compare $P(T_r, i)$ and $P(B_r, i)$ pairwise for each r and thus exclude influence of the starting population p_r on the results.

In our case studies we considered the five tactics presented in Section 3 with the following weights and thresholds:

- **Component reallocation:** The threshold for high utilisation is $\Delta_{\text{realloc}} = 0.4$. The weight W_{realloc} is 1.0.
- **Faster hardware:** The threshold for high utilisation is $U_{\text{faster}} = 0.75$. The increase factor f is 25%. The weight W_{faster} is 0.1.
- **More hardware:** The threshold for high utilisation is $U_{\text{more}} = 0.8$. The weight W_{more} is 0.5.
- **Slower Hardware:** The threshold for low utilisation is $U_{\text{slower}} = 0.25$. The decrease factor is 25%. The weight W_{slower} is 0.1.
- **Consolidate Server:** The threshold for low utilisation is $U_{\text{cons}} = 0.3$. The weight W_{cons} is 1.

For the performance prediction, we configured the LQN-

[1]This metric definition is not valid if run T performed worse that run B. However, this never happened in our analyses, thus we do not complicate the metric to cover it.

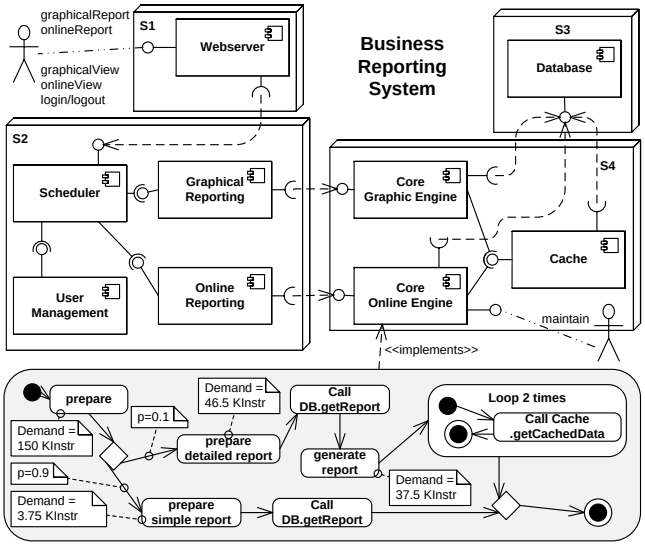

Figure 5: PCM model of the Business Reporting System

Solver with convergence value 0.001, iteration limit 50 and underrelaxation coefficient 0.5 (c.f. [11]).

4.3 Case I: Business Reporting System

For the first case study, we analysed the so-called business reporting system (BRS). It allows users to retrieve statistical reports about business processes from a database and is based on a real system [29].

Fig. 5 shows a condensed excerpt of the initial PCM model of the BRS visualised using annotated UML diagrams[2]. The components are initially allocated to four different servers. Besides components and servers, the PCM model of the BRS contains the behaviour model (cf. Sec. 2). Fig. 5 shows an example behaviour model for the CoreOnlineEngine component in the lower part.

The case study analyses an open workload usage scenario [22], with an arrival rate of 0.5 users per second. Users access multiple services of the system per interaction.

The BRS server costs depend on the chosen CPU processing rate pr. For the costs model, we analysed Intel's CPU price list [13]. We fitted a power function to this data, so that the resulting costs of one server s is $cost_s = 0.7665\ pr_s^{6.2539} + 145$ with coefficient of determination $R^2 = 0.965$. The overall server costs of one candidate is the sum of the costs of all used servers. The server costs of the initial system are 618 units. The performance prediction shows that the system is overloaded, i.e. its queue lengths grow over time and operational equilibrium is not reached [14].

All components of the system can be freely allocated to different servers to spread the load and improve performance. The processing rates pr_s of each server s can be varied continuously between 0.75 and 3 GHz. With 9 components in the BRS system and up to nine used servers, we obtain 18 degrees of freedom: 9 allocation degrees and 9 server sizing degrees. In this case study, the considered objectives are mean user response time and server costs.

[2]The complete model is described in more detail at http://sdqweb.ipd.kit.edu/wiki/PerOpteryx.

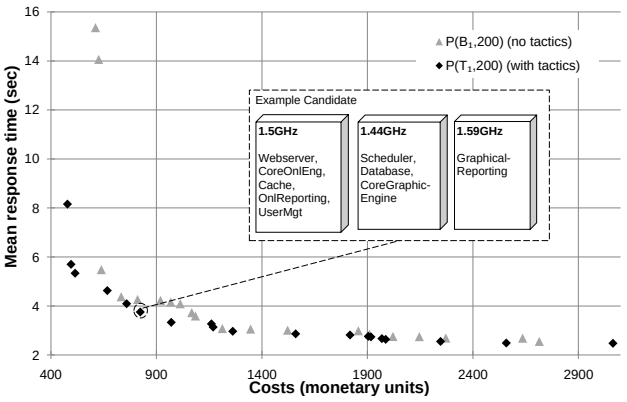

Figure 6: Pareto Fronts (Performance vs. Costs) of the Business Reporting System showing an Advantage for the Optimisation Run with Tactics ($i = 200$)

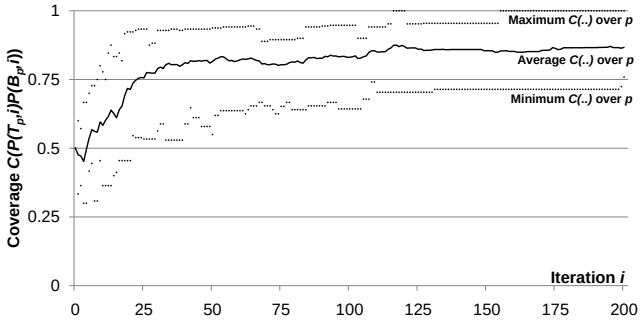

Figure 7: Pareto front coverage $C(P(T_r, i), P(B_r, i))$ of runs using tactics T over unguided runs B for $r \in 0, ..., 9$ (BRS system)

Fig. 6 shows the comparison of two Pareto fronts generated by optimization runs T_1 and B_1 at iteration 200 as an example. Recall that the initial candidate had infinite response time and costs of 618 units. The runs generated a number of Pareto-optimal candidates with a mean response time between 2.4 and 15.4 seconds and costs between 479 and 3063 units. One concrete Pareto-optimal candidate is circled in Fig. 6 as an example. Compared to the initial system, it has an improved response time of 3.76 sec at slightly higher costs of 824. It uses only 3 servers with higher processing rates, a better allocation of the components, and the resource-intensive "Graphical Reporting" component is allocated on a dedicated server. Some other candidates on the left even have lower costs and better response time than the initial system, which is achieved by better allocation and adjusted processing rates.

The optimization run using tactics T_1 yielded 19 Pareto-optimal candidates, which dominate 18 of the 21 candidates found by the unguided optimization run B_1. None of the results from T_1 is dominated by B_1. This results in a coverage metric of $C(P(T_1, 200), P(B_1, 200)) = 19/22 = 0.86 > 0.5$. From the figure, we notice that the tactics have improved the Pareto front for low costs values, i.e. on the left side of the graph, particularly well. Here, the distance between the fronts is largest.

Fig. 7 shows the minimum, average, and maximum coverage $C(P(T_r, i), P(B_r, i))$ over all 10 starting populations p_r

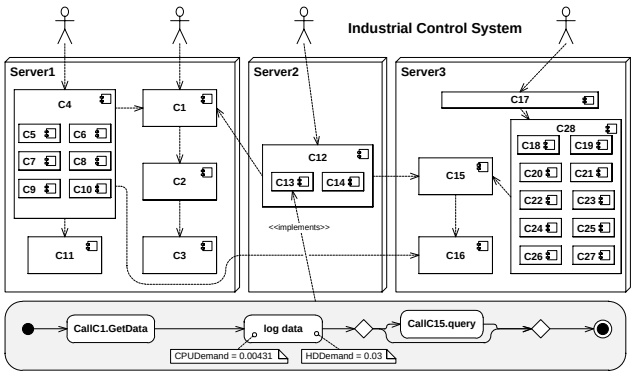

Figure 8: PCM model of the Industrial Control System

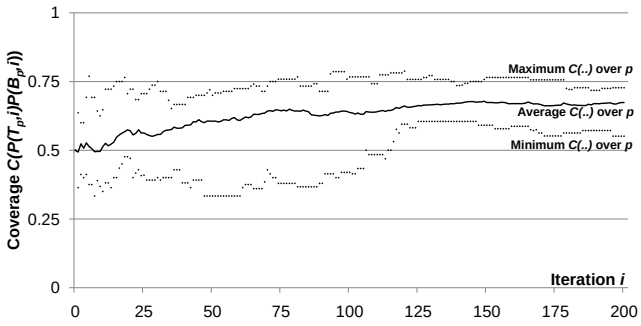

Figure 9: Pareto front coverage $C(P(T_r, i), P(B_r, i))$ of runs using tactics T over unguided runs B for $r \in 0, ..., 9$ (ABB system)

for each iteration step i. We observe that the average coverage is larger than 0.5 starting from few iterations, rapidly increases to a value of around 0.8 at iteration 33, and then increases more slowly up to an average value of 0.86. Even the worst-performing run using tactics, i.e. the minimum coverage, is larger than 0.5 for iteration 20 and later. Student's t-test confirms that the average coverage $C(P(T_r, i), P(B_r, i))$ is significantly larger than 0.5 for all $i > 16$ with a significance level $\alpha = 0.99$.

Concerning the speed-up, the optimization run with tactics was able to find an equivalent front 153 iterations earlier than the optimization without tactics on average. Thus, for our formerly defined metric, we get $D = 80$, meaning an 80% improvement in runtime. The speed-up is statistically significant with a lower confidence interval bound of 70% improvement in runtime ($\alpha = 0.99$).

4.4 Case II: Industrial Control System

The second case study provides additional evidence about the supposed improvement of the guided search and also shows the applicability of the method in an industrial context on a large scale system. In this case, we analysed an industrial control system from ABB, which is used in many domains, such as power generation, pulp and paper handling, and oil and gas processing. It comprises of several million lines of C++ code.

Fig. 8 shows a part of the PCM model of the system. We have modelled 28 components of the system, each one having at least one resource demand, which were determined from performance measurements on a running instance of the system. The resource environment is adaptable to customer requirements and consists of three servers in our initial configuration. For the hardware resources, we used a costs model similar to the former case study. One behaviour model for component `C13` is shown in Fig. 8 at the lower part. Additionally, we modelled four of the most important usage scenarios of the system.

Our optimization aimed at lowering the costs for the overall system while keeping the response time of one usage scenario within tolerable limits. As degrees of freedoms, it is possible to replace component `C1` and `C13` by alternative implementations with different performance and costs. Furthermore, the allocation of the components to hardware resources can be adjusted and the processing rates of the servers can be lowered to save costs. As in the former study, we again generated 10 starting populations and ran both the unguided and tactic-guided variant of PerOpteryx.

In total PerOpteryx found on average 33 Pareto-optimal candidates per run[3]. One example candidate had its costs reduced by 23.1%, while the response time increased by 19.4% which is tolerable within customer requirements. For this candidate PerOpteryx suggested to use the standard variants of components `C1` and `C13`, to purchase a slightly more powerful CPU for server 1, and then to deploy all components on this server, so that the others servers can be removed to save costs. Indeed this candidate reflects a realistic configuration of the system that is sold to smaller customers.

Again, we study the development of coverage metric C as the search advances in Fig. 9. In this case, we observe that the average coverage is again larger than 0.5 starting from few iterations and increases to a value of around 0.67 at iteration 142, and then stays at that level until iteration 200. This time, the worst-performing run using tactics, i.e. the minimum coverage, is inferior to its unguided counterpart until iteration 117, but then also improves to values larger than 0.5. Student's t-test confirms that the average coverage $C(P(T_r, i), P(B_r, i))$ is significantly larger than 0.5 for all $i > 61$ ($\alpha = 0.99$).

In case II, the optimization runs with tactics were able to find an equivalent front 108 iterations earlier than their counterpart without tactics on average. Thus, for our formerly defined metric, we get $D = 56$, meaning an 56% improvement in runtime. The speed-up is statistically significant with a lower confidence interval bound of 46% improvement in runtime ($\alpha = 0.99$). We also noted that all optimization runs with tactics found more Pareto-optimal candidates than their counterpart without tactics.

4.5 Discussion

The case studies showed that PerOpteryx can successfully determine a meaningful Pareto front in both cases, which enable the software architect to weigh up the quality attributes and choose one candidate. Significant performance improvements and/or costs savings are possible compared to the initial design. Additionally, the tactics can significantly speed-up the search process (up to 80% in our cases). Assuming a fixed time given for the optimization, as it is often the case in industrial settings, the tactics lead to significantly better results (candidates with lower costs and better performance).

[3]The Pareto fronts can be found at sdqweb.ipd.kit.edu/wiki/PerOpteryx/Tactics_Case_Study

Notice that conflicting tactics (e.g. scaling out and consolidating) are not detrimental to the search process: The search should develop the population in both directions. We weighted the tactics so that cost-neutral performance tactics, such as reallocation, are preferred. PerOpteryx can be used to analyse three or more quality attributes and multiple degrees of freedom [20].

Some limitations are inherent to our approach. It inherits the limitations of the underlying performance predictions techniques (PCM/LQN) [4]. Additionally, because the underlying performance models are expressive, the evaluation of each candidate takes several seconds. Thus, our approach can hardly be applied at runtime to reconfigure systems if immediate reactions are expected. Furthermore, our approach does not guarantee to find the real Pareto front, i.e. the globally optimal solutions, because metaheuristics are used [8].

5. RELATED WORK

Our work is based on software performance prediction [27, 2, 17] and multi-objective metaheuristic optimisation [8].

Rule-based Approaches: Xu et al. [30] present a semi-automated approach to find configuration and design improvements on the model level. Based on a LQN model, performance problems (e.g., bottlenecks, long paths) are identified in a first step. Then, mitigation rules are applied.

Diaz-Pace et al. [10] have developed the ArchE framework. ArchE assists the software architect during the design to create architectures that meet quality requirements. It provides the evaluation tools for modifiability or performance analysis, and suggests modifiability improvements.

All rule-based approaches share two common limitations. The model can only be changed as defined by the improvement rules. However, as especially performance is a complex and cross-cutting quality attribute, optimal solutions could lie on search paths not accessible by rules. Additionally, the rules only improve a single quality attribute, so other quality attributes are likely to degrade and the approaches cannot determine the optimal trade-offs (i.e. the Pareto-optimal candidates, see Section 2.2) well.

Specialised Optimisation Approaches (for software architectures or more general) on analytic quality models have been suggested for a defined set of degrees of freedom (such as e.g. component selection and component allocation [15]) in a certain environment under a set of assumptions. However, these approaches only consider the given degree of how to change the system, thus, they cannot find an overall optimal solution in all available degrees of freedom. Works from the area of self-adaptive systems [6] tackle a different optimisation problem that focusses on quickly finding a satisfiable solution, also considering adaptation costs, instead of finding the optimal trade-offs to support human design decisions as required at design time.

Metaheuristic-based Approaches: Aleti et al.[1] present a generic framework to optimize architectural models with evolutionary algorithms for multiple arbitrary quality attributes. As a single degree of freedom, they vary the deployment of components to hardware nodes. Canfora et al. [5] optimize service composition costs using evolutionary algorithms while satisfying SLA constraints. Only service selection is considered as a degree of freedom.

Menascé et al. [23] generate service-oriented architectures that satisfy quality requirements, using service selection and architectural patterns. They use random-restart hill-climbing. The only supported degrees of freedom are the introduction of two architectural patterns, namely load balancing and one fault tolerance mechanism, the hardware level cannot be considered.

All metaheuristic-based approaches to software architecture improvement explore only one or few degrees of freedom of the architectural model. Our approach offers multiple degrees of freedom, such as component allocation, hardware configuration, and component selection, and is extendible for more by plugging in additional model transformations. In addition, except for [1], the other approaches do not enable trade-off decisions because they assume fixed quality requirements or a given utility function, and then optimize only a single quality attribute or utility function. In contrast, PerOpteryx can optimize an arbitrary number of quality attributes independently. In this work, we consider performance and costs, while PerOpteryx also supports reliability [20].

Additionally, like our preliminary work [20], all these approaches do not use domain-specific knowledge (e.g. tactics) to guide the search. For the related domain of software-intensive system design, Grunske et al. [12] survey more related optimization approaches. None of the metaheuristic approaches uses additional domain-specific knowledge to guide the search.

Problem-specific Knowledge in Metaheuristic Search: In the field of metaheuristic search techniques [8], problem-specific knowledge can be integrated into a metaheuristic in several ways [7]. First, the problem representation itself contains knowledge about the domain. For example, genetic encoding can be chosen so that only feasible solutions are constructed. In this work, the encoding only allows valid architectures.

Second, the performance of the search can be enhanced by problem-specific knowledge. For example, Cheng et al. [7] present a heuristic crossover operator based on a problem-specific neighbourhood definition. So far, these heuristic operators are defined based on static properties of the search problem. In this work, we suggest to use detailed domain-specific rules (as used in the rule-based approaches) in a new type of heuristic operator. To the best of our knowledge, this kind of heuristic operators and the resulting hybrid optimization has not been described before.

6. CONCLUSIONS

We have presented the PerOpteryx approach for improvement of software architecture models using a metaheuristic search guided by architectural tactics. We have implemented a selection of performance and costs tactics and integrated them into the metaheuristic search process of PerOpteryx. Two case studies on industry-sized systems validated our approach and demonstrated the potential for improving quality attributes.

Both practitioners and research can benefit from our approach. *Practitioners* gain a tool for automatic improvement software architecture designs in an efficient way. The tool assists them in overcoming the craftsmanship-like trial-and-error approach to software architecture design as it makes known performance and costs rules accessible to them. The result can be systems with better performance, which are built according to engineering principles. *Researchers* get a demonstration for the combination of rule-based and meta-

heuristic approaches applied to software architecture model improvement. The included concepts are not PerOpteryx-specific, but could be included into other design optimization approaches, such as ArcheOpteryx [1] and SASSy [23]. The generic list of tactics can serve as a template for future approaches and be extended to other quality attributes.

In future work, we aim at extending PerOpteryx to incorporate reliability tactics [20]. We plan to take uncertainties for model parameters into account when executing the predictions.

Acknowledgements

The authors thank Tom Beyer for implementing the initial tactics for PerOpteryx and contributing valuable ideas.

7. REFERENCES

[1] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of AADL models. *Proc. of the ICSE Workshop on MOMPES*, pages 61–71, 2009.

[2] 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.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley, 2003.

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

[5] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qoS-aware service composition based on genetic algorithms. In *Proc. of GECCo*, pages 1069–1075. ACM, 2005.

[6] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, Berlin, Germany, 2009. 10.1007/978-3-642-02161-9_1.

[7] R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms. II. Hybrid genetic search strategies. *Computers & Industrial Eng.*, 37(1-2):51–55, 1999.

[8] C. A. Coello Coello, C. Dhaenens, and L. Jourdan. Multi-objective combinatorial optimization: Problematic and context. In *Advances in Multi-Objective Nature Inspired Computing*, volume 272 of *Studies in Computational Intelligence*, pages 1–21. Springer, 2010.

[9] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, volume 1917/2000, pages 849–858. Springer, 2000.

[10] A. Díaz Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating quality-attribute reasoning frameworks in the archE design assistant. In S. Becker, F. Plasil, and R. Reussner, editors, *QoSA*, volume 5281 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2008.

[11] G. Franks, T. Omari, C. M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. on Software Engineering*, 35(2):148–161, 2009.

[12] L. Grunske, P. A. Lindsay, E. Bondarev, Y. Papadopoulos, and D. Parker. An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In *WADS*, volume 4615 of *LNCS*, pages 188–209. Springer, 2006.

[13] Intel Corporation. Intel®processor price list, effective feb 8th, 2010. http://www.intc.com/priceList.cfm, 2010. last visit March 10th, 2010.

[14] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[15] T. Kichkaylo, A.-A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *Proc. of IPDPS*, pages 3–3, Los Alamitos, CA, Apr. 22–26 2003. IEEE Computer Society.

[16] E. Kornyshova and C. Salinesi. MCDM Techniques Selection Approaches: State of the Art. In *IEEE Symp. on Computational Intelligence in Multicriteria Decision Making*, pages 22–29, April 2007.

[17] H. Koziolek. Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation*, 67(8):634–658, 2010.

[18] H. Koziolek and R. Reussner. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *Proc. of SIPEW Workshop*, volume 5119 of *LNCS*, pages 58–78. Springer, 2008.

[19] M. Kuperberg, K. Krogmann, and R. Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proc. of CBSE*, volume 5282 of *LNCS*, pages 48–63. Springer, October 2008.

[20] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically improve software models for performance, reliability and cost using genetic algorithms. In *Proc. 1st Joint WOSP/SIPEW Int. Conf. on Performance Engineering (ICPE'10)*, pages 105–116. ACM, 2010.

[21] A. Martens, H. Koziolek, L. Prechelt, and R. Reussner. From monolithic to component-based performance evaluation of software architectures. *Empirical Software Engineering*, 2011. To appear.

[22] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

[23] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, and J. a. P. Sousa. A framework for utility-based service oriented design in SASSY. In *Proc. 1st Joint WOSP/SIPEW Int. Conf. Conference on Performance Engineering (ICPE'10)*, pages 27–36. ACM, 2010.

[24] Microsoft Cooperation. *Improving .NET Application Performance and Scalability (Patterns and Practices)*. Microsoft Press, 2004.

[25] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06), 2006.

[26] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, Mar. 2008.

[27] C. U. Smith and L. G. Williams. *Performance Solutions*. Addison-Wesley, 2002.

[28] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[29] X. Wu and M. Woodside. Performance Modeling from Software Components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.

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

[31] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Trans. on Evolutionary Computation*, 7:117–132, 2002.