Integrating Statistical Response Time Models in Architectural Performance Models

Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim von Kistowski, Samuel Kounev
University of Würzburg, Germany
{firstname}.{lastname}@uni-wuerzburg.de

***Abstract*— Performance predictions enable software architects to optimize the performance of a software system early in the development cycle. Architectural performance models and statistical response time models are commonly used to derive these performance predictions. However, both methods have significant downsides: Statistical response time models can only predict scenarios for which training data is available, making the prediction of previously unseen system configurations infeasible. In contrast, the time required to simulate an architectural performance model increases exponentially with both system size and level of modeling detail, making the analysis of large, detailed models challenging. Existing approaches use statistical response time models in architectural performance models to avoid modeling subsystems that are difficult or time-consuming to model, yet they do not consider simulation time.**

In this paper, we propose to model software systems using classical queuing theory and statistical response time models in parallel. This approach allows users to tailor the model for each analysis run, based on the performed adaptations and the requested performance metrics. Our approach enables faster model solution compared to traditional performance models while retaining their ability to predict previously unseen scenarios. In our experiments we observed speedups of up to 94.8%, making the analysis of much larger and more detailed systems feasible.

***Keywords*-Component-based systems, Architectural performance models, Statistical response time models**

I. Introduction

Performance predictions of software systems allow software architects to evaluate the performance of different architecture alternatives [1]. A common approach to predict the performance of software systems are architectural performance models [2]. These models describe the architecture of a system, its control flow and its performance relevant parameters. Discrete event simulation of these models allows predicting performance metrics for the modeled system, such as resource utilizations or response times.

The time required to simulate a model is an important factor for performance models. Nambiar et al. [3] identify faster model solution as a criterion for performance modeling success, Woodside et al. [4] find that reduced run-time would increase the adoption of simulation-based solvers and Koziolek et al. [5] state that the time required to solve a model is a limiting factor for the level of detail a performance model can contain.

Existing approaches for the simplification of performance models either focus on making the model more understandable for users or are limited to analytical solvers [6, 7, 8, 9, 10]. A number of approaches have been proposed to build statistical response time models that predict the performance of a software system by training machine learning models on observations of the system's performance [11, 12, 13, 14]. These statistical response time models provide fast predictions for the impact of changes in workload intensity and workload parameterization but struggle to predict the impact of changes to the underlying system architecture or its deployment accurately. However, no existing approach enables faster solution of architectural performance models while retaining the prediction accuracy and capability to predict previously unseen scenarios.

In this paper, we introduce a generic modeling approach that enables a parallel and integrated description of subsystems as statistical response time models and as traditional queuing models in order to speed up model simulation. The proposed approach allows users to dynamically select the appropriate modeling composition of statistical and queuing sub-models for each analysis run. We provide a transformation of the integrated queuing/statistical model to Queueing Petri Net (QPN) [15] and extend an existing discrete event simulation solver for QPNs to support statistical response time models. Additionally, we investigate the impact of replacing components or full subsystems by statistical response time models on the architectural performance model's accuracy and simulation time.

The approach presented in this paper enables faster solution of architectural performance models while retaining the prediction accuracy and capability to predict previously unseen scenarios of traditional, queuing theory-based performance models. This enables software architects to analyze larger systems, performance engineers benefit from the ability to build more detailed models and self-adaptive systems can explore additional adaptation options within the same time period due to the faster model solution. The approach presented in this paper represents a step towards solving a problem that impedes widespread adoption of performance models in practice: models that are either too large or too detailed cannot be simulated within a reasonable time frame [3, 4, 5].

In our evaluation, we apply the proposed approach to a distributed, component-based system of medium size. Our experiments show that the approach maintains sufficient prediction accuracy and achieves speedups of up to 94.8%.

II. Approach

In this paper, we propose a generic meta-modeling approach to extend component-based performance models with the ability to describe a component or subsystems as a statistical response time model in parallel to the traditional description (Section II-A). We develop an algorithm to extract statistical response time models from monitoring data (Section II-B) and introduce an approach to transform architectural performance models containing statistical response time models to an extended QPN (Section II-C).

A. Modeling

In general, our modeling approach can be applied to any architectural software performance model which uses a component notation. Examples for software performance models which could be extended using our approach include the Palladio Component Model (PCM) [16], Component-Based Modeling Language (CBML) [17], Prediction-Enabled Component Technology (PECT) [18], or Components with Quantitative properties and Adaptivity (COMQUAD) [19] as all of these models rely on the notion of repository components with some kind of performance description which are instantiated as assembly components. The meta-model proposed in this paper would not be directly applicable to performance models, such as ROBOCOP [20], which do not differentiate between repository and assembly components. However, the general concept of including statistical response time models in architectural performance models is also applicable to these performance models, but would require a different modeling approach.

We show how our approach can be applied to the Descartes Modeling Language (DML) [21], a performance model for component-based systems in data centers. It is representative for component-based performance models and was already evaluated in a number of case studies [22, 23, 24, 25]. The meta-model of DML is separated into six submodels as shown in Figure 1:

- **Application architecture**
  The application architecture meta-model consists of a component repository and a component assembly. The repository specifies the interfaces and components of a system. For each component the performance-relevant properties can be specified via resource demands, control flow operations (loops, branches, etc.) and calls to interface providing roles. In the assembly, these components are instantiated and connected to each other according to their interface providing and requiring roles.
- **Resource landscape** The physical resources in a data center such as compute or storage nodes, are contained in the resource landscape meta-model. The DML provides support for virtualized environments, such as Virtual Machiness (VMs) or containers as nested resources.

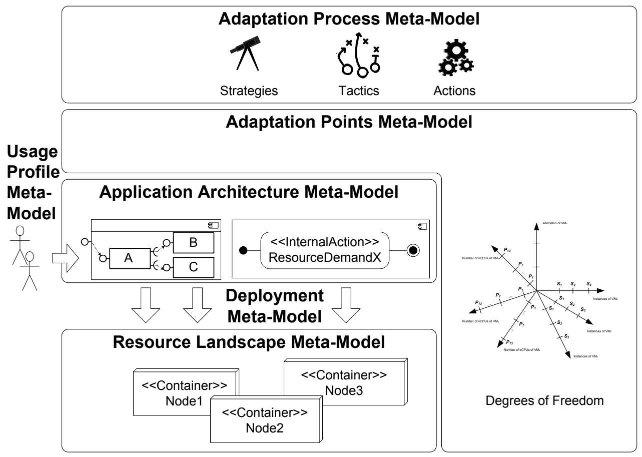

Adaptation Process Meta-Model
Strategies
Tactics
Actions
Adaptation Points Meta-Model
Usage Profile Meta-Model
Application Architecture Meta-Model
A
B
C
<<InternalAction>> ResourceDemandX
Deployment Meta-Model
Resource Landscape Meta-Model
<<Container>> Node1
<<Container>> Node2
<<Container>> Node3
Degrees of Freedom

Figure 1: Structure of the DML meta-model (source: [22])

- **Deployment** The deployment meta-model maps the instantiated assembly components from the application architecture meta-model to the physical resources defined in the resource landscape meta-model.
- **Usage profile** The workload is specified in the usage profile meta-model, similar to UML use cases and UML activities. Here, DML supports open and closed workloads, as well as detailed user sessions.
- **Adaptation points** The possible adaptations to a system at run-time are limited, as not everything can be changed during system operation. Therefore, the adaptation points meta-model describes which elements of the resource landscape and application architecture can be adapted at run-time.
- **Adaptation process** The adaptation process meta-model describes how a dynamic system reacts if its environment changes during operation. The DML supports three granularities which allow building composite adaptation processes: strategies, tactics, and actions.

Component assembly in DML is modeled similarly to Unified Modeling Language (UML), as shown in Figure 2. Every `RepositoryComponent` is either a `BasicComponent`, a `CompositeComponent` or a `SubSystem`. Every `RepositoryComponent` specifies a number of `InterfaceProviding-` and `InterfaceRequiringRoles`, which describe the functionality a component provides and what functionality should be provided by external components. Any `RepositoryComponent` can be instantiated multiple times as `AssemblyContexts`. These component/subsystem instances are connected with each other by `AssemblyConnectors`, which connect `InterfaceProvidingRoles` to `InterfaceRequiringRoles`.

Traditionally, performance descriptions are modeled on the level of repository components to enable reuse of the performance descriptions [16, 17, 18, 19]. However, sta-

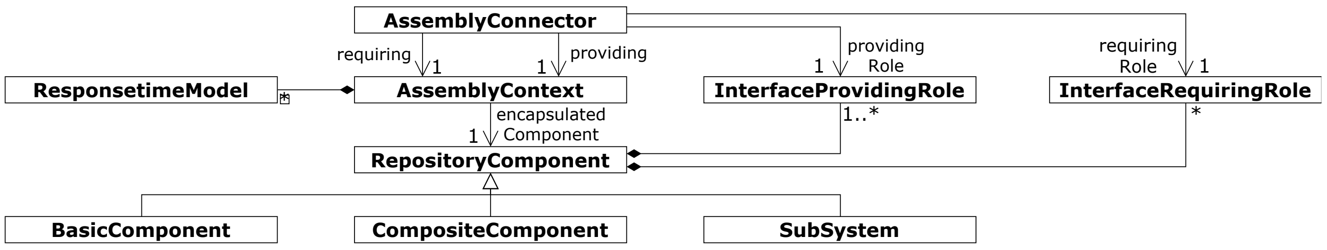


Figure 2: Meta-model for DML assembly with response time model integration

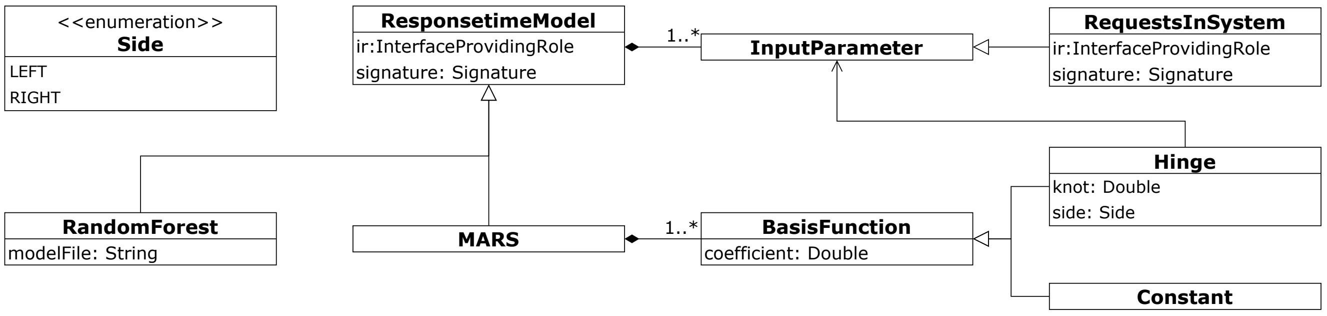


Figure 3: Meta-model for response time models

tistical response time models describe the response time of a specific component instance and the response time of a component is influenced by its required services and deployment [5]. Therefore, we propose to enrich component instances in the form of `AssemblyContexts` with `StatisticalModels` describing its response time. This allows replacing component instances, composite component instances, and subsystem instances with statistical response time models during model solution.

Every response time model describes the response time for a specific workload class, as shown in Figure 3. In DML, a workload class can be uniquely identified by an `InterfaceProvidingRole` and a corresponding `Signature`. Therefore, every `Responsetime-Model` references an `InterfaceProvidingRole` and a `Signature`. Every `ResponsetimeModel` has a number of `InputParameters` based on which the response time for a specific request is calculated. Currently, the only supported `InputParameter` is `RequestsInSystem`. Upon arrival of a new request for which a response time has to be calculated, this parameters counts how many requests of a workload class (specified by an `Interface-ProvidingRole` and a `Signature`) are currently being processed by the component/subsystem. The `Input-Parameter` interface provides an extension point to include further parameters that influence the response time of a component/subsystem, e.g., workload parameters such as the size of an input file. Response time models can either be a white-box, i.e., a human readable function or a black-box, i.e., a machine learning model. We model one of each, in order to showcase how they can be integrated into the model. As a black-box approach, we use a trained `Random-Forest` model. Every `RandomForest` references a file containing the machine learning model. This model predicts the response time of a component/subsystem by using the `InputParameters` as features. `MARS` is a regression technique that results in a human-readable function, which can therefore be modeled explicitly. Every `MARS` model consists of a sum of `BasisFunctions`, which are either a static `Constant` or a `Hinge`. A `Hinge` $H(i, k, c, s)$ is a function over an `InputParameter` i with the following form:

$$H(i, k, c, s) = \begin{cases} c * max(0, k - i) & \text{for } s = LEFT, \\ c * max(0, i - k) & \text{else} \end{cases} \quad (1)$$

with a constant c, a knot k and a side s. For further information on random forests or MARS see [26] and [27], respectively.

Our modeling approach provides three key characteristics: (i) It provides an extension point to include any regression or machine learning model. Although deep learning approaches might often be unfeasible due to the required quantity of training data, including them would be possible. (ii) Our approach annotates component/subsystem instances with response time models. The response time of a component/subsystem depends on the deployment platform and the required services, which means the response time, unlike resource demands, cannot be specified for generic components. (iii) The presented meta-model enables parallel modeling of a component/subsystem as a traditional queuing system and as a response model. As the appropriate modeling granularity depends on the predicted performance metrics and model adaptations, this enables adapting the model for each individual request.

B. Extraction

Response time models can be extracted from either run-time monitoring or via dedicated measurements. There is a large body of work on the construction of response time models using dedicated measurements, e.g., [12, 13, 14]. This research usually focuses on the intelligent selection of measurement points.

In the following, we present an approach to extract response time models from run-time monitoring. We assume generic monitoring data that consists of a set monitoring records $rec = (wc, st, cp)$, where wc denotes the request's workload class, st and cp represent the absolute start and completion time of the request. Training regression or machine learning models requires a set of observations of a target variable and a number of features which will be used to predict the target variable. In our case, the target variable is the response time of a single request and the features are the number of requests of each workload class that are currently being processed within the component/subsystem. Therefore, an observation $obs = (rt, wc, req_{wc_1}, req_{wc_2}, ...)$ consists of the observed response time rt, the workload class of the processed request wc and the number of requests being processed in the component/subsystem upon arrival of the request for every workload class req_{wc_i}.

Algorithm 1 shows how we extract the training data for the regression and machine learning approaches based on run-time monitoring data. The algorithm receives a list of records $recordList$ as input and returns a list of observations $observations$. It maintains a list of requests that are currently being processed in the component/subsystem $requestsInSystem$ and a map $wcToCount$ that counts the number of requests currently being processed for each workload class in order to speed up the computation. In line 5, the algorithm iterates over the list of monitoring records (we assume that this list is ordered by request arrival time). For every monitoring record, an observation is created in lines 12-16. The observation consists of the response time of the request $record.cp - record.st$, the workload class of the request $record.wc$ and the number of requests already in the system for each workload class $wcToCount.get(wc_i)$. Afterwards, the request is added to the list of requests currently in the system $requestsInSystem$ and the counter for the number of requests currently in the system $wcToCount$ is increased for the workload class of the request.

Prior to this, the algorithm checks if any of the requests currently in the system has departed prior to the arrival of the current request in lines 6-11. It iterates over every request currently in the system and evaluates if the completion

Algorithm 1 Training data extraction algorithm

```
1: function EXTRACTTRAININGDATA($recordList$)
2:     $observations$ = {}
3:     $requestsInSystem$ = {}
4:     $wcToCount = \{wc_1 \rightarrow 0, wc_2 \rightarrow 0, ...\}$
5:     for $record$ in $recordList$ do
6:         for $request$ in $requestsInSystem$ do
7:             if $request.cp < record.st$ then
8:                 $requestsInSystem$.remove($request$)
9:                 $wcToCount$.decrease($request.wc$)
10:            end if
11:        end for
12:        $observations$.add(new Observation(
13:                        $record.cp - record.st$,
14:                        $record.wc$,
15:                        $wcToCount$.get($wc_1$),
16:                        $wcToCount$.get($wc_2$), ...))
17:        $requestsInSystem$.add($request$)
18:        $wcToCount$.increase($request.wc$)
19:    end for
20:    return $observations$
21: end function
```

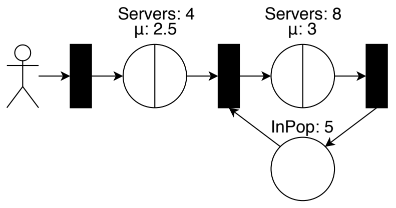
timestamp of the request $request.cp$ is smaller and therefore prior to the start time stamp of the new record $record.st$. Every request that finishes is removed from the list of requests currently in the system $requestsInSystem$ and the counter for the number of requests currently in the system $wcToCount$ is decreased for the corresponding workload class.

The presented algorithm converts common run-time monitoring data that consists only of start and completion time for each request and its request class (which in case of a REST interface would be the queried URL) to a format which enables training of most machine learning approaches and the fitting of stochastic models. The corresponding processes to train a Multivariate Adaptive Regression Splines (MARS) or random forest model on this data is described in [26] and [27], respectively.

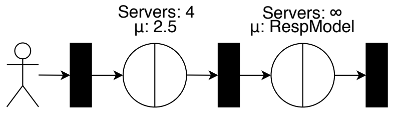
C. Model solution

Traditionally, architectural performance models are transformed into a solution formalism such as a Queueing Network (QN), Layered Queueing Network (LQN) or Queueing Petri Net (QPN). DML, the modeling formalism used in this work, derives performance predictions by transforming the model to a QPN that is subsequently simulated using simQPN [15]. For detailed information on the transformation to QPN we refer to [28].

We extend the existing formalism by introducing a new distribution for the processing time, where the processing time is calculated by a statistical response time model based on the number of requests already being served. As the



(a) Initial queuing model



(b) Queueing model with response time model

Figure 4: Example for the integration of statistical response time models in queuing models.

statistical response time model already considers queuing time, the corresponding queuing place has infinite servers to avoid duplicating the queuing time. It is important to note that each statistical response time model only describes the response time for a single request class. Therefore, a queuing place that serves two request classes has to contain two response time models.

Figure 4 shows an example how a subsystem can be replaced by a statistical response time model. In Figure 4a the original queuing model is depicted. It consists of two queues, which represent two components and a place with five initial tokens, that is used to model a limited software thread pool of five for the second component. Figure 4b shows the system if the second component is replaced by a statistical response time model. The full representation of the component (the queuing place and the ordinary place for the software thread pool) is replaced by a single queuing place with infinite servers and a processing rate which is described by a statistical response time model. This example describes the replacement of a single, coarsely modeled component. For more complex components or subsystems, the full model describing the component/subsystem is still replaced by a single queue with infinite servers and a statistical response time model. Assuming the statistical response time model perfectly predicts the response time of the subsystem/component, we can assume that the model with the statistical response time model predicts the same values for the following performance indices as the original model:

- Throughput of the model
- Overall response time of the model
- Utilization of all remaining queues

As the derivation of a single value from a response time model is significantly faster than the simulation of a request through a subsystem, the overall model solution time is greatly reduced.

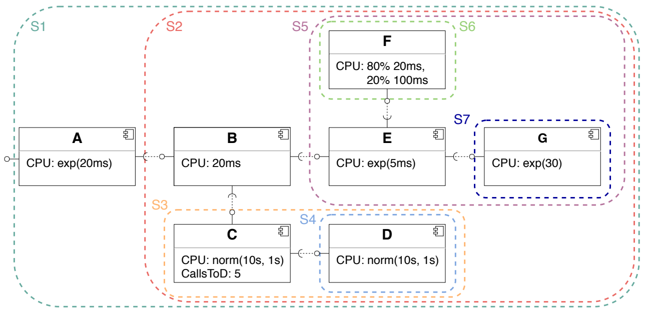


Figure 5: System consisting of components A-Z with their performance properties and the different models sections S1-S7 that can be replaced by statistical response time models.

III. Case Study

We design our case study in order to answer the following three research questions:

- **RQ1:** Does replacing parts of the performance model with statistical response time models decrease prediction accuracy?
- **RQ2:** What are limiting factors for applying statistical response time models?
- **RQ3:** How much can the integration of statistical models decrease the time required to simulate a system?

Based on these research questions, we analyze a medium sized, distributed system with seven components featuring diverse performance properties. We construct a traditional performance model for this system and extract statistical response time models as described in Section II-B. Next, we compare the prediction accuracy of the traditional performance model to using different statistical response time models. Finally, we analyze how much applying different statistical response time models speeds up model solution.

Experiment Setup: For this case study, we deploy a system composed of components with synthetic resource demands on seven virtual machines with two 2.6 GHz cores and 4 GB memory, each. The virtual machines are deployed in a CloudStack cluster (version 4.9 with KVM) consisting of eight HPE ProLiant DL160 Gen9 hosts with eight 2.6 GHz cores and 32 GB ram each. Hyper-threading was disabled on all hosts to avoid race conditions. The architecture of the system is shown in Figure 5. It consists of seven components with constant, exponentially distributed and normal distributed resource demands. Additionally, component C contains a loop that calls component D five times and component F contains branches where its resource demand is 20 ms 80% of the time and 100 ms otherwise. For our experiments, we use the load driver introduced in [29].

Model Construction: First, we build a traditional performance model using DML, based on the resource demands shown in Figure 5. After some initial analysis, we added network delays of about 3 ms to each external call, which

Load	Measured / Predicted Utilization [%]						
[Req/s]	A	B	C	D	E	F	G
10	11.4 / 9.9	11.3 / 10.0	6.6 / 5.0	25.7 / 25.0	4.0 / 2.5	18.0 / 18.0	15.3 / 15.0
15	15.9 / 15.0	16.0 / 14.9	9.6 / 7.5	37.3 / 37.4	5.4 / 3.7	26.2 / 26.9	22.2 / 22.5
20	20.7 / 20.1	21.2 / 20.1	12.8 / 10.1	48.9 / 50.2	6.7 / 5.1	34.7 / 36.2	28.6 / 30.1
25	25.6 / 25.1	25.8 / 25.1	15.3 / 12.6	60.6 / 62.8	7.9 / 6.3	42.9 / 45.3	36.1 / 37.6
30	30.2 / 29.9	30.4 / 29.9	11.7 / 15.0	72.7 / 74.7	9.4 / 7.5	50.9 / 54.1	42.8 / 42.5
35	35.3 / 35.0	35.3 / 35.0	20.8 / 17.5	88.5 / 87.8	10.9 / 8.8	60.0 / 63.0	50.5 / 52.4
40	38.5 / 40.0	39.1 / 40.0	24.2 / 20.0	96.7 / 99.9	12.2 / 10.0	67.9 / 72.0	55.3 / 59.9

Table I: Comparison of measured utilizations of components A-G and the predictions by the queuing theory model.

Load	Measured / Predicted Responsetime [ms]						
[Req/s]	A	B	C	D	E	F	G
10	226 / 209	200 / 185	93 / 81	16 / 14	83 / 82	42 / 40	34 / 34
15	229 / 215	204 / 192	95 / 85	16 / 14	85 / 84	43 / 41	35 / 35
20	236 / 226	211 / 202	99 / 92	17 / 16	87 / 87	45 / 43	35 / 36
25	254 / 245	228 / 221	112 / 106	20 / 19	92 / 92	47 / 47	38 / 37
30	283 / 281	257 / 256	136 / 132	25 / 24	96 / 99	49 / 51	40 / 40
35	339 / 391	312 / 365	184 / 232	34 / 44	102 / 109	54 / 58	42 / 43
40	4805 / 21614	4778 / 21587	4632 / 21438	924 / 4287	120 / 124	68 / 68	45 / 47

Table II: Comparison of measured responsetimes of components A-G and the predictions by the queuing theory model.

Statistical	Predicted Utilization [%]						
Models	A	B	C	D	E	F	G
None	25.1	25.1	12.6	62.8	6.3	45.3	37.6
S1	0	0	0	0	0	0	0
S2	25.0	0	0	0	0	0	0
S3	25.0	25.0	0	0	6.3	45.1	37.5
S4	25.0	25.0	12.5	0	6.2	44.9	37.5
S5	25.0	25.0	12.5	62.5	0	0	0
S6	25.0	25.0	12.5	62.5	6.3	0	37.5
S7	25.0	25.0	12.5	62.6	6.3	45.1	0
S3 + S5	25.0	25.0	0	0	0	0	0
S3 + S6	25.0	25.0	0	0	6.3	0	37.5
S3 + S7	24.9	25.0	0	0	6.2	45.0	0
S4 + S5	25.0	25.0	12.5	0	0	0	0
Measured	25.6	25.8	15.3	60.6	7.9	42.9	36.1

Table III: Predicted utilization at 25 req/s when applying different statistical models (see Figure 5) and the measured values as baseline.

Statistical	Predicted Responsetime [ms]						
Models	A	B	C	D	E	F	G
None	245	221	106	19	92	47	37
S1	266	-	-	-	-	-	-
S2	263	239	-	-	-	-	-
S3	265	240	131	-	86	42	36
S4	284	260	145	26	91	46	37
S5	253	228	105	18	99	-	-
S6	252	227	105	18	98	55	35
S7	252	227	106	19	98	46	44
S3 + S5	270	245	130	-	95	-	-
S3 + S6	269	244	131	-	90	48	34
S3 + S7	270	245	130	-	91	42	41
S4 + S5	295	267	145	26	99	-	-
Measured	254	228	112	20	92	47	38

Table IV: Predicted responsetime at 25 req/s when applying different statistical models (see Figure 5) and the measured values as baseline.

we model as resource demands on a resource with infinite servers, as the network delays appear to be static. In order to collect monitoring data to construct the statistical response time models, we put the system under varying loads, ranging from 10 requests per second to 50 requests per second for ten minutes. Next, we use the algorithm presented in Section II-B to generate a set of training data to train a MARS model on. For the MARS model we use the py-earth python package[1] with the following parameters:

- minspan_alpha = 5
- endspan_alpha = 5
- sample_weight = 1 + 1/(concurrency + 1)

These parameters are derived based on systematic experimentation to optimize the internal GCV error score of the MARS model, which describes how well the model fits the training data. The sample weights prioritize lower concurrency levels in order to decrease the influence of measurements during overload scenarios. Using this approach, we extract a total of seven statistical response time models, shown in Figure 5 as S1-S7. These response time models have varying size. For example, S1 replaces the full system with a statistical response time model whereas S4 only replaces a single component. We integrate these response time models into the DML model as described in Section II-A.

Prediction accuracy: In a first step, we validate the prediction accuracy of the traditional performance model to ensure its validity. We ran seven experiments with a constant load of 10-40 requests per second for three minutes and measured the utilization and response time for every

[1]https://contrib.scikit-learn.org/py-earth/content.html

component. Table I compares the measured utilization to the predictions of the traditional queuing theory model. Most predictions are accurate with an error of less than two percentage points. For components C, D, F, and G the prediction is slightly off in the high load scenarios, but still always less than five percentage points. Table II shows the measured response times along the predicted values. Here, the prediction error is $\leq$10% for the experiments with 10-30 requests per second. For 35 requests per second, the model overestimates the system response time but remains below the 30% considered sufficient for capacity planning [1]. At 40 requests per second, the response time predictions are inaccurate, which is expected as the system is under excess load, so there is no steady-state for the response time. Overall, the accuracy of the model seems sufficient and will be our comparison values for the following experiments.

Next, we apply the different statistical response time models S1-S7 as well as some combinations of them (S3 + S5, S3 + S6, S3 + S7, S4 + S5) and compare the resulting prediction accuracy to the traditional performance model without statistical response time models (”None”). Table III shows the resulting utilization predictions. We performed this analysis for all load levels, but show only the results for 25 requests per second due to space constraints. The full dataset is available online[2]. The first observation here is that the model predicts a utilization of zero for any component which is replaced by a statistical model. This can be explained by the fact, that the statistical models only predict a response time, but do not schedule any resource demands. Therefore, the first limitation for the integration of statistical models in performance models is that we can not replace components or subsystems that are deployed on physical resources of which we want to predict the utilization (**RQ2**). The average prediction accuracy across all utilization predictions without statistical models is 9.2%$\pm$9.4, compared to 9.1%$\pm$9.4 for the average prediction accuracy across all utilization predictions using models containing statistical response time models. Therefore, we conclude that the utilization predictions remain accurate (**RQ1**). Table IV shows the same data for the response time predictions. Here, when we replace a component with a statistical model, we lose the ability to predict the response time of any components called by this component as the statistical response time model does not contain any notion of external calls. This is a conceptual limitation for the use of statistical response time models within performance models (**RQ2**). The response time predictions are in some cases more accurate than their queuing model counterparts and worse in other cases. The average prediction accuracy across all response time predictions without statistical models is 6.3%$\pm$6.4, compared to 12.6%$\pm$20.6 for the average prediction accuracy across all response time predictions using

[2]https://github.com/SimonEismann/ICSA2019/blob/master/results.xlsx

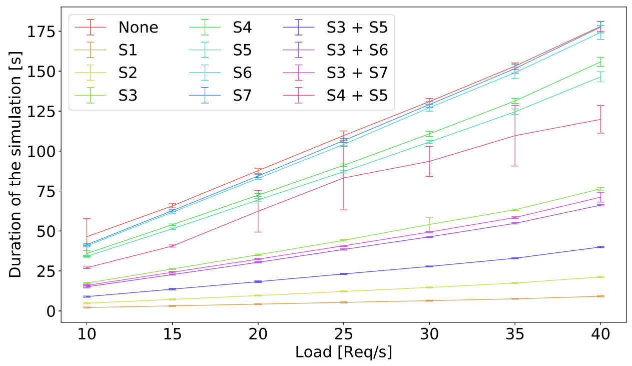


Figure 6: Wall clock time required to simulate 200.000 seconds of simulated time when applying different statistical models (see Figure 5).

models containing statistical response time models. While the prediction accuracy deteriorates slightly, it remains overall satisfactory (**RQ1**).

It is interesting to note here, that the prediction accuracy is not tied to the size of the system that is replaced by a statistical response time model. For example, the S4 + S5 scenario replaces four components and achieves the worst prediction accuracy, whereas S1 replaces seven components but results in better prediction accuracy. Instead, the prediction accuracy seems to be tied to how well the MARS model is able to fit the response time data, i.e., the better the MARS model predicts the monitoring data, the more accurate the overall predictions become. Here, the investigation of other machine learning approaches such as random forests would be of interest. Overall, replacing parts of the performance model seems to not invalidate the performance predictions, but to make the prediction of some performance metrics impossible (**RQ1** & **RQ2**).

Simulation time analysis: The previous experiment shows that replacing parts of the performance model makes the prediction of some performance indices impossible, but does not negatively impact the accuracy of the remaining predictions. In the next step, we analyze the speedup that can be gained in these scenarios. In order to measure the simulation time, we measure the wall clock time [30] required to simulate a total of 200 000 seconds for each scenario from the previous experiment. The measurements were performed on an out-of-the-box Lenovo X1 Yoga Gen 2 Thinkpad with an i7-7600U CPU with up to 2.80 GHz. As there is some variation in the results, we repeat the measurements 20 times each and calculate confidence intervals.

Figure 6 shows the required wall clock time to simulate 200 000 seconds for load levels from 10 to 40 requests per second. The required time increases linearly with the number of requests per second. This behavior is expected, as doubling the number of requests per second roughly doubles the number of events that need to be processed within the event-based simulation. The model without any statistical response time models is the slowest and takes up to 178 seconds. The fastest is S1 with an average simulation time of 9.1 seconds for 40 requests per second, resulting in a speedup of 94.8%. On the other hand, S6 and S7 result in almost no speed up, as they only replace a single component with a statistical response time model. Curiously, S4 also replaces only a single component but still speeds the simulation up by ~60%. The difference here seems to be that component D is called five times as often as components E/F. So we can conclude that the speedup depends on the number of calls to the replaced components. Based on S1-S7 we can observe that the achieved speedup is also related to the size of the system which is replaced by a statistical response time model. Lastly, the results for models containing multiple statistical response time models (S3+S5, S3+S6, S3+S7 and S4+S5) indicate that applying multiple models increases the speedup compared to their singular counterparts (S3, S4, S5, S6, S7). Overall, this experiment shows that integrating statistical response time models can significantly decrease the time required to simulate a system (**RQ3**).

IV. Limitations & Threats to Validity

While our approach shows significant benefits over the current state of the art, there still are limitations and threats to the validity to be discussed.

Currently, the statistical response time models are trained based on the observed concurrency levels for each request class. However, the literature suggests that the parameterization of requests within a single request class can significantly impact resource demands [5, 31]. Our approach could be extended to take this into account by having these parameters implement the *InputParameter* interface shown in Figure 3.

Most components are stateless, but for a stateful component, the response time can be influenced by its internal state [32]. The approach presented in this paper is currently not able to capture this behavior accurately. In theory, response time models could be able to accurately capture such behavior if data on previous requests is included in the training data and a machine learning technique which can handle non-linear correlations is used. However, this is outside the scope of this paper.

The experimentation in this paper considers a restricted scenario as there is no co-location of components on the same physical resources and the only investigated adaptation is the load level. Further experimentation including scenarios with co-located components, adaptations to the system architecture and changing component implementations is required to derive a definitive rule set describing when statistical response time models can safely be integrated in architectural performance models. Overall, a case-study using a realistic benchmarking application (i.e., TeaStore [33]) will be part of our future work.

Lastly, the exact results for the attained speedups by the integration of statistical response time models in architectural performance models are limited to MARS models. Further experimentation using different machine learning algorithms to build statistical response time models could provide additional insights. However, for most machine learning algorithms the training time is the limiting factor and deriving predictions from a previously trained model is comparatively fast [34].

V. Related Work

The existing work related to this paper can be divided into three groups: Integration of measurement-based approaches in performance models, statistical performance models, and model reduction techniques.

Integration of measurement-based approaches in performance models: Performance models are usually parameterized based on estimations from experts, dedicated performance experiments or run-time monitoring data. However, few approaches provide an explicit integration of measurement-based approaches in performance models.

The behavior of modern storage systems is difficult to explicitly model. Therefore, Noorshams et al. [35] propose a methodology to enrich software architecture modeling approaches with statistical I/O performance models. This approach enables accurate performance predictions for software systems using dedicated storage systems, without having to explicitly model them. This approach is limited to storage systems and does not allow to dynamically switch between the statistical model and a traditional queuing model for the storage system.

Woodside et al. [36] propose the concept of performance-related completions, where an abstract model element is later replaced with a concrete model, once the information is available. Happe et al. [37] extend this approach by using platform independent model skeletons. The platform-specific details are derived based on measurements from automatically generated test drivers. This approach incorporates measurements in performance models in order to improve the parameterization of the performance model and therefore the performance prediction accuracy.

To the best of our knowledge, no existing approach allows to dynamically switch between statistical response time models and queuing model representations for generic components or sub-systems.

Statistical performance models: Statistical performance models are also known as software performance curves [14, 38], performance prediction functions [12], performance predictions using machine learning [39] or performance predictions using statistical techniques [11]. These approaches train machine learning/statistical models on measurement data, which is usually collected during dedicated measurements. These models are used to infer the performance of a software system with different workloads or configurations.

Thereska et al. [11] build a statistical performance model to predict the performance of several Microsoft applications, such as the Office suite or Visual Studio. By instrumenting the applications, the authors collect data from several hundred thousand real users. Classification and Regression Trees (CART) is used to filter relevant features, followed by a similarity search to derive performance predictions.

Kwon et al. [39] use a regression model to predict the performance of Android applications to determine whether the task can be efficiently offloaded. The authors train the regression model not only on the input parameters and the current hardware utilization but also on values calculated during the program execution.

Westermann et al. [12] compare four techniques for the construction of statistical performance models: MARS, CART, Genetic programming, and Kriging. In their case studies, MARS significantly outperformed the other three approaches.

Noorshams et al. [13] evaluate the accuracy of regression techniques for the performance prediction of storage systems: linear regression, MARS, CART, M5 Trees and Cubist Forests. The authors propose an approach to optimize the parameterization of the individual algorithms. With parameter optimization MARS and Cubist outperform CART, M5 Trees and the linear regression in their case study.

Faber et al. [14] use genetic programming to derive software performance curves. They introduce parameter optimization approaches and a technique to prevent overfitting for genetic programming. In their evaluation, the optimized genetic programming approach outperforms an unoptimized MARS model.

Summarizing, we can say that statistical performance models can predict the impact of changes in workload intensity and workload parameterization well, but are unreliable when predicting the impact of changes to the system or its deployment. Based on the results reported in these previous studies, we use MARS in our approach as a representative for statistical performance models.

Performance model reduction: Various techniques to reduce or simplify performance models have been proposed. Some techniques aim to improve the model solution speed whereas others attempt to make the model easier to understand for human users.

Queuing networks can be simplified by replacing a number of nodes with a so-called Flow-Equivalent Server (FES) [6, 7]. A FES is a load-dependent queue, which perfectly emulates the delay caused by a subsystem. The resulting network can be solved faster using analytical approaches, such as convolution or Mean Value Analysis (MVA). However, FES come with two drawbacks: i) the representation as a load-dependent queue is cumbersome for discrete-event-simulation and ii) in order to construct a FES the short-circuited network needs to be solved once for every possible number of concurrent users, so in order to construct a FES for a system with 1000 users, the model needs to be solved a thousand times.

The method of surrogate delays by Jacobson et al. [8] enables an analytical solution for models with simultaneous resource possession. The approach requires a model, where the primary resource is estimated by a delay and another one where the secondary resource is estimated by a delay. While this method does reduce the initial model, it requires an additional model and does not provide any advantages when solved using simulation.

A recent approach by Islam et al. [9] reduces the size of LQNs by aggregating activities, tasks, and entries of non-bottleneck resources. This approach is extended by an improved approach for the identification of tasks that can be safely aggregated in [10]. As this approach permanently reduces the model, it is no longer possible to accurately analyze the impact of reconfigurations or deployment changes for non-bottleneck resources.

To the best of our knowledge, there is no existing model reduction approach for simulation-based solvers that enables faster model solution while retaining the full flexibility and accuracy of the initial model.

VI. Conclusion

Architectural performance models provide a potent tool for software architects to evaluate and improve the performance of a software system and its architecture. However, the time required to simulate large or detailed models has been identified as a limiting factor by many researchers [3, 4, 5]. This paper aims to solve this by integrating statistical response time models into architectural performance models. The appropriate model composition depends on the performance metrics of interest. Therefore, our approach enables modeling of each component and subsystem as a queuing network and a statistical response time model in parallel. This allows to dynamically tailor the system description for each analysis run.

Our approach enables software architects to analyze larger systems, performance engineers can explore more detailed models and self-adaptive systems can explore additional adaptation options within the same time period due to the faster model solution. For a mid-sized system with a distributed, component-based architecture, our approach achieves speedups of up to 94.8% while maintaining sufficient prediction accuracy. As future work, we are looking to automate the model tailoring step to integrate this approach in our vision for self-aware performance models [40].

VII. Acknowledgments

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] D. A. Menasce and A. F. A. Virgilio, *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*, 1st ed. PTR, 2000.

[2] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press, 2016.

[3] M. Nambiar, A. Kattepur, G. Bhaskaran, R. Singhal, and S. Duttagupta, "Model driven software performance engineering: Current challenges and way ahead," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 4, pp. 53–62, 2016.

[4] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, 2007, pp. 171–187.

[5] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.

[6] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, 1998.

[7] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[8] P. A. Jacobson and E. D. Lazowska, "The method of surrogate delays: Simultaneous resource possession in analytic models of computer systems," *SIGMETRICS*, vol. 10, no. 3, pp. 165–174, Sep. 1981.

[9] F. Islam, D. Petriu, and M. Woodside, "Simplifying layered queuing network models," in *European Workshop on Performance Engineering*. Cham: Springer International Publishing, 2015, pp. 65–79.

[10] ——, "Choice of aggregation groups for layered performance model simplification," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, USA: ACM, 2018, pp. 241–252.

[11] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, pp. 1–12, 2010.

[12] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated inference of goal-oriented performance prediction functions," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 190–199.

[13] Q. Noorshams, D. Bruhn, S. Kounev, and R. Reussner, "Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, USA: ACM, 2013, pp. 283–294.

[14] M. Faber and J. Happe, "Systematic adoption of genetic programming for deriving software performance curves," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 33–44.

[15] S. Kounev and A. Buchmann, "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation," *Performance Evaluation*, vol. 63, no. 4-5, pp. 364–394, 5 2006.

[16] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.

[17] X. Wu and M. Woodside, "Performance modeling from software components," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 290–301, Jan. 2004.

[18] S. A. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau, "Packaging predictable assembly," in *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, ser. CD '02. London, UK: Springer-Verlag, 2002, pp. 108–124.

[19] S. Göbel, C. Pohl, S. Röttger, and S. Zschaler, "The comquad component model: Enabling dynamic selection of implementations by weaving non-functional aspects," in *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, ser. AOSD '04. New York, USA: ACM, 2004, pp. 74–82.

[20] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, "Modelling of input-parameter dependency for performance predictions of component-based embedded systems," in *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. EUROMICRO '05. IEEE, 2005, pp. 36–43.

[21] F. Brosig, N. Huber, and S. Kounev, "Architecture-Level Software Performance Abstractions for Online Performance Prediction," *Elsevier Science of Computer Programming Journal (SciCo)*, vol. 90 Part B, pp. 71–92, 2014.

[22] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 432–452, 2017.

[23] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures," *IEEE Computer*, vol. 49, no. 7, pp. 53–61, 2016.

[24] S. Eismann, J. Walter, J. von Kistowski, and S. Kounev, "Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 135–144.

[25] S. Spinner, J. Grohmann, S. Eismann, and S. Kounev, "Online model learning for self-aware computing infrastructures," *Journal of Systems and Software*, vol. 147, pp. 1 – 16, 2019.

[26] T. K. Ho, "Random decision forests," in *Document analysis and recognition, 1995., proceedings of the third international conference on*, vol. 1. IEEE, 1995, pp. 278–282.

[27] J. H. Friedman, "Multivariate adaptive regression splines," *The annals of statistics*, pp. 1–67, 1991.

[28] F. Brosig, "Architecture-level software performance models for online performance prediction," Ph.D. dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014.

[29] J. von Kistowski, M. Deffner, and S. Kounev, "Run-time Prediction of Power Consumption for Component Deployments," in *Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC 2018)*, September 2018.

[30] R. Fujimoto, "Parallel and distributed simulation," in *Proceedings of the 2015 Winter Simulation Conference*, ser. WSC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–59.

[31] V. Ackermann, J. Grohmann, S. Eismann, and S. Kounev, "Black-box learning of parametric dependencies for performance models," in *Proceedings of the 13th International Workshop on Models@run.time (MRT 2018)*, October 2018.

[32] L. Happe, B. Buhnova, and R. Reussner, "Stateful component-based performance models," *Software and Systems Modeling*, vol. 13, no. 4, pp. 1319–1343, 2014.

[33] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2018)*, September 2018.

[34] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

[35] Q. Noorshams, R. Reeb, A. Rentschler, S. Kounev, and R. Reussner, "Enriching software architecture models with statistical models for performance prediction in modern storage environments," in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '14. ACM, 2014, pp. 45–54.

[36] M. Woodside, D. Petriu, and K. Siddiqui, "Performance-related completions for software specifications," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, USA: ACM, 2002, pp. 22–32.

[37] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric performance completions for model-driven performance prediction," *Perform. Eval.*, vol. 67, no. 8, pp. 694–716, Aug. 2010.

[38] D. Westermann and C. Momm, "Using software performance curves for dependable and cost-efficient service hosting," in *Proceedings of the 2Nd International Workshop on the Quality of Service-Oriented Software Systems*, ser. QUASOSS '10, 2010, pp. 3:1–3:6.

[39] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *Proceedings of the 2013 USENIX Annual Technical Conference*. Berkeley, USA: USENIX Association, 2013, pp. 297–308.

[40] J. Grohmann, S. Eismann, and S. Kounev, "The Vision of Self-Aware Performance Models," in *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, April 2018, pp. 60–63.