An Industrial Case Study of Performance and Cost Design Space Exploration

Thijmen de Gooijer
Industrial Software Systems
ABB Corporate Research
Västerås, Sweden
thijmen@acm.org

Anton Jansen
Industrial Software Systems
ABB Corporate Research
Västerås, Sweden
anton.jansen@se.abb.com

Heiko Koziolek
Industrial Software Systems
ABB Corporate Research
Ladenburg, Germany
heiko.koziolek@de.abb.com

Anne Koziolek
Department of Informatics,
University of Zurich
Zurich, Switzerland
koziolek@ifi.uzh.ch

ABSTRACT

Determining the trade-off between performance and costs of a distributed software system is important as it enables fulfilling performance requirements in a cost-efficient way. The large amount of design alternatives for such systems often leads software architects to select a suboptimal solution, which may either waste resources or cannot cope with future workloads. Recently, several approaches have appeared to assist software architects with this design task. In this paper, we present a case study applying one of these approaches, i.e. PerOpteryx, to explore the design space of an existing industrial distributed software system from ABB. To facilitate the design exploration, we created a highly detailed performance and cost model, which was instrumental in determining a cost-efficient architecture solution using an evolutionary algorithm. The case study demonstrates the capabilities of various modern performance modeling tools and a design space exploration tool in an industrial setting, provides lessons learned, and helps other software architects in solving similar problems.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.11 [**Software Engineering**]: Software Architecture

1. INTRODUCTION

Evolving a software intensive system is typically far from trivial. One of the first steps in this process is to create a common shared vision among the system stakeholders for the future of the system. Once this vision has been established, a system road-map can be created that outlines the steps and time-schedule in which the system should evolve. However, creating a reasonable vision and associated road-map proves to be complicated in practice. Often, the trade-offs among the quality attributes are not understood well enough to make an informed decision. One way to improve this understanding is by performing design space exploration. In such an exploration, quantitative analysis models are created that evaluate various architectural alternatives with respect to the system's relevant quality attributes. In turn, this activity creates a deeper understanding of the trade-offs among the quality attributes, thereby enabling more informed decision making.

A challenge for the aforementioned approach is that its associated methods and tools are largely untested in an industrial setting outside the academic research groups they originated from. This creates uncertainty about whether these methods and tools are fit for purpose and actually deliver the value they promise. This in turn stands in the way of popularization, i.e. the ability of an approach to gain wide spread industrial acceptance [38].

The main contribution of this paper is therefore a case study presenting the application of various academic tools and methods for design space exploration in an industrial setting. Our case study presents how we explore component re-allocation, replication, and hardware changes and their performance and cost implications. To the best of our knowledge, this combination of explored changes has not been automatically explored for performance in other works yet. We present the selection criteria for the used methods and tools, the application of them, and the results they delivered. Finally, we present lessons learned and provide pointers for future research directions.

The rest of this paper is organized as follows. Section 2 introduces the system under study, the performance and costs goals of the case study, and the overall approach followed. Next, Section 3 presents the performance measurements, which are used in Section 4 to build a performance model. Section 5 reports on our manual exploration of the design space with the aforementioned performance model, the formalization of our cost model, and how we used both to automatically explore the degrees of freedom in our design space. Lessons learned and pointers for future research directions are presented in Section 6. The paper concludes

with related work in Section 7 and with conclusions and future work in Section 8.

2. CASE STUDY OVERVIEW

2.1 System under study

The system studied in this paper is one of ABB's remote diagnostic solutions (RDS). The RDS is a 150 kLOC system used for service activities on thousands of industrial devices and records device status information, failures, and other data. We note that throughout the paper certain details of the system are intentionally changed to protect ABB's intellectual property.

During normal operation the industrial devices periodically contact the RDS to upload diagnostic status information. In cases of abnormal behavior, the devices upload error information to the RDS for future analysis. Customers can track the status of their devices on a website and can generate reports, for example, showing device failures over the last year. Service engineers can troubleshoot device problems either on-site or remotely by sending commands to the device through the RDS.

Part of the RDS is illustrated in Fig. 2. The devices run device specific software that connects to the 'RDS Connection Point', which runs in ABB's DMZ (perimeter network for security reasons). Here the data enters ABB's internal network and is send onward to the core components on the application server.

The system core components handle both the processing and storing of the uploaded data, as well as the publishing of data and interaction with external systems. Data that is received from devices is processed and then stored in the database. Certain data uploads are mined in the 'Data Mining and Prediction Computation' component, for example, to predict the wear of parts. The customer website is hosted outside the RDS back-end and gets data from the RDS web services via a proxy (not shown). The website for service engineers is hosted within the same environment as the RDS web services. Both websites offer access to reports that are created by a separate reporting component, which is not shown in the model.

The RDS is connected to various other systems. One example is shown in the diagram in Fig. 2: the 'ABB customer and device database' interface, which represents a Microsoft SQL Server (MS-SQL) plug-in that synchronizes the RDS database against a central ABB database recording information on which customers have what service contracts for which devices. This synchronization scheme reduces the latency for look-up of this information when a human user or device connects to the system.

2.2 Performance and Cost Goal

ABB wants to improve the performance of RDS by re-architecting, because its back-end is operating at its performance and scalability limits. Performance tuning or short term fixes (e.g., faster CPUs) will not sustainably solve the problems in the long term for three reasons. Firstly, the architecture was conceived in a setting where time-to-market took priority over performance and scalability requirements. Hence, the current architecture has not been designed with performance and scalability in mind. Secondly, the number of devices connected to the back-end is expected to grow by an order of magnitude within the coming years. Finally, the amount of data that has to be processed for each device, is expected to increase by an order of magnitude in the same period. Together, these dimensions of growth will significantly increase the demands on computational power and storage capacity.

The performance metric of main interest to the system stakeholders is the device upload throughput, i.e., the number of uploads the system can handle per second. It was decided that the system resource on average must not be utilized more than 50% to be able to cope with workload peaks. Considering that the speed of the target hardware resources will grow significantly in the next years, the performance goal for the system was specified as: "The system resources must not be utilized more than 50 percent for a ten times higher arrival rate of device uploads".

The architectural redesign should manage to fulfill the performance goal while controlling cost at the same time. It is not feasible to identify the best design option by prototyping or measurements. Changes to the existing system would be required to take measurements, but the cost and effort required to alter the system solely for performance tests are too high because of its complexity. Furthermore, the capacity predicted by a performance model can be combined with the business growth scenario to get a time-line on the architectural road-map. Thereby, we can avoid the risk of starting work too late and experiencing capacity problems, or being too early and making unnecessary investments. Therefore, ABB decided to create a performance model and cost model to aid architectural and business decision making, to conduct capacity planning and to search the design space for architectural solutions that can fulfill the performance goal in a cost effective manner.

2.3 Case Study Activities

Our case study consisted of three major activities: performance measurement (Section 3), performance modeling (Section 4), and design space exploration (Section 5). Fig. 1 provides an overview of the steps performed for the case study. The following sections will detail each step.

3. PERFORMANCE MEASUREMENT

Measurements are needed to create an accurate performance model. To accurately measure the performance of the RDS, a number of steps needs to be performed. First, tools have to be selected (Section 3.1). Second, a model should be created of the system workload (Section 3.2). Finally, measurements have to be performed (Section 3.3).

3.1 Measurement Tool Selection

The first step entails finding the appropriate tools needed to measure the performance of the system. In short, this consists of:

- A load generator tool to simulate stimuli to the systems in a controlled way.
- An Application Performance Management (APM) tool, which can measure the response time of different stimuli (called business transactions) to the system.
- A (distributed) profiler, which can tell us how the response time of the different stimuli is distributed. This information is vital, as we would like to understand how the performance is build up to focus our re-architecting efforts.

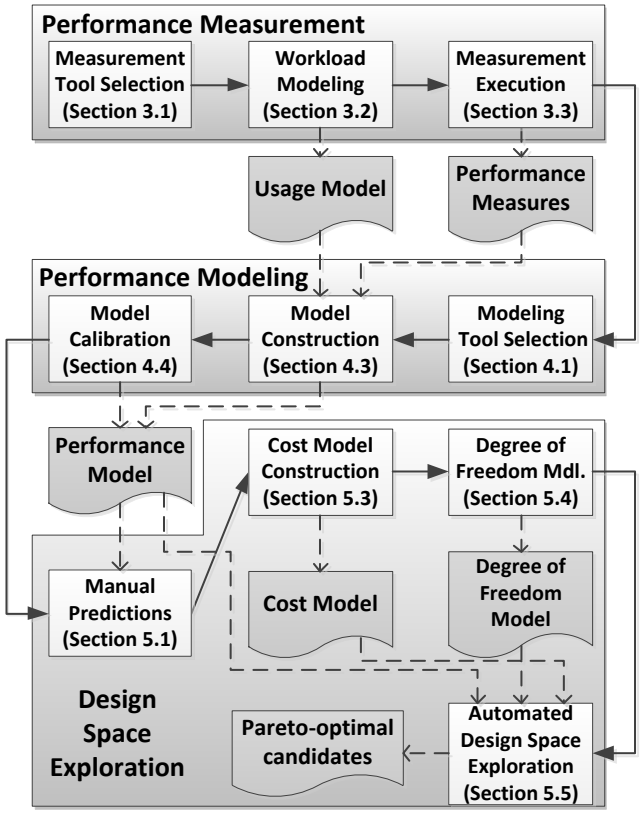

Figure 1: Case Study Approach

We created an initial list of 58 different tools that could fulfill some of this functionality. After removing the alternatives that were no longer maintained or lacked feature completeness the list shrunk to 17 tools. For each of these 17 tools, we classified their functionality and costs by attending online sales presentations of the various tool vendors. In the end, we settled on using the NeoLoad load generator tool [17] in combination with the dynaTrace APM tool [16], because the tools integrate nicely, dynaTrace makes instrumentation of .Net applications easy and NeoLoad supports MS Silverlight.

The dynaTrace tool offers normal performance measurement functionality and distributed profiling functionality. DynaTrace traces requests through all tiers in a distributed application and stores measurements on each individual request in so-called PurePaths. DynaTrace instruments the .NET application code in the CLR layer, thus allowing PurePaths to show timings (i.e., CPU time, execution time, latency) as deep as at the method-level. The recorded measurements can be summarized and viewed in several ways. For example, dynaTrace can create sequence diagrams of PurePaths or show a break-down of how much time was spent in various APIs.

3.2 Workload Modeling

The second step deals with the issue of finding out what the typical workload on the system is. Firstly, we organized a workshop with the developers to find out the actors on the system and their main use cases. Secondly, we turned on the logging facilities of the IIS containers to record the stimuli to the system for a month in production. Using the Sawmill log analysis tool [18], we determined the most frequent used use cases: the periodic uploading of diagnostic/error information by devices and the interaction of Service Engineers (SE) with the system. Surprisingly enough, the customer related use cases were relatively low in frequency. Most likely this is due to customers being only interested in interacting with the system when the devices have considerable issues, which is not often the case.

The RDS thus executes two performance-critical usage scenarios during production: periodic uploading of diagnostic status information from devices and the interaction of service engineers (SE) with the system. We approximated the uploads with an open workload having an arrival rate of 78.6 requests per minute. Furthermore, we characterized the SE interactions with a closed workload with a user population of 39.3 and a think time of 15 seconds. All values were derived from the production logs of the system.

We decided to run load tests with the system on three different workload intensities: low, medium, and high. The workloads are specified in Table 1 as the number of sustained uploads received from the devices per minute, and the number of concurrent service engineers requesting internal web pages from the system.

The medium workload approximates the current production load on RDS. The low workload was used as an initial calibration point for the performance model and is approximately half the production load. The advantage of the low workload is that the system behavior is more stable and consistent, making it easier to study. The high workload represents a step towards the target capacity and enables us to study how the resource demands change at increasing loads.

workload	uploads/min	SE requests/min
low	41.0	20.5
medium	78.6	39.3
high	187.9	93.9

Table 1: The model calibration workloads used. (data is altered to protect ABB's intellectual property)

3.3 Measurement Execution

The third and final step, performing the measurements, has to deal with an important constraint to the case study: the need to minimize the impact of the study on ongoing development and operation activities of the system. To address this issue, we built a separate "experimental" copy of the system in the ABB Corporate Research labs. This copy consisted of a recently released version of the RDS, which is deployed on a large server running virtualization software. This deployment in virtual servers allows us to easily test out different deployments of the system with varying hardware resources. For the virtualization software we choose to go with VMWare ESX, as we have local in-house IT expertise to manage such servers.

The experimental copy of the RDS runs on three virtual machines. The NeoLoad load generator runs on a separate physical machine to emulate industrial devices uploading data and service engineers generating requests to the RDS. DynaTrace data collection agents were installed on the DMZ and application server. Information on the performance of the database server was recorded by subscribing dynaTrace to its Windows performance monitor counters, as dynaTrace cannot instrument the Microsoft SQL Server (MS-SQL).

During the first load tests on our system, we verified the consistency of the performance measurements and we gained sufficient confidence in dynaTrace's instrumentation to run all our measurements for 30 minutes. In the next tests, we stressed the system to observe its behavior under peak loads and to find its bottlenecks and capacity limits. Both test phases needed several iterations to adjust dynaTrace's instrumentation, so that requests were traced through all tiers correctly. During the stress tests we varied the hardware configuration of the virtual machines to explore the sensitivity of the application to the amount of CPU cores and memory and several concurrency settings of the ASP.Net container.

Finally, we performed two types of structured measurements to support the performance modeling. First, we ran load tests matching our workload model, which later could be compared to model predictions to calibrate the model. We used average values from these load tests to instatiate our model. Second, we measured just a single request to get a clear picture of runtime system behavior to base the behavioral part of the performance model upon. When recreating the workload model in NeoLoad, we needed several iterations until the generated workload matched the model.

Some data we could not gather using dynaTrace. First of all, some metrics were not easily recorded or isolated. For example, network latency measurements were more easily obtained using a ping tool and MS-SQL's performance counters were better studied with the MS-SQL profiler tool. Second, it was difficult to interpret results. For example, significant differences between CPU and execution time were difficult to account for, because the instrumentation of the ASP.Net container itself was insufficient.

4. PERFORMANCE MODEL

To construct a performance model for the ABB RDS, we first selected an appropriate modeling notation (Section 4.1), which turned out to be the Palladio Component Model (Section 4.2). Based on the performance measurements results, the workload model, and additional analyses, we constructed a Palladio model for the ABB RDS (Section 4.3), which we calibrated (Section 4.4) until it reflected the performance of the system under study well.

4.1 Method and Tool Selection

We conducted a survey of performance modeling tools [23] and selected initial candidates based on three criteria: (i) support for performance modeling of software architectures, (ii) available recent tooling, (iii) tool maturity and stability. Most mature tools do not meet the first criterion, while prototypical academic tools often fail on the latter two as described in the following.

The low-level Petri Net modeling tools GreatSPN [15] and ORIS [7] as well as the SHARPE [12] tool do not reflect software architectures naturally. This problem also applies to PRISM [10]. The ArgoSPE [1], and TwoTowers [14] tools are not actively updated anymore. Intermediate modeling languages, such as KlaperSuite [4] or CSM [2], were discarded due to their still instable transformation from UML models.

The commercial Hyperformix tool [35] is expensive, while from publicly available information it is difficult to judge whether the tool offers major benefits in our use case. The PEPA tools [9] use a powerful process algebra, but the prototypical mapping from UML models to PEPA has not been maintained for several years.

Six tools appeared mature enough and promising to fit our architectural modeling problem: Java Modeling Tools (JMT) [3], the Layered Queuing Network Solver (LQNS) [5], Palladio workbench [8], QPME [11], SPE-ED [13], and Möbius [6].

Based on our requirements, we analyzed the different tools. Some of Möbius' formalisms do not target at software architecture performance modeling. The SPE-ED tool specifically targets architecture modeling, but it is no longer actively updated. Furthermore, both Möbius and SPE-ED are subject to license fees for commercial use. While their price is reasonable, the acquisition of commercial software within a corporation considerably delays work. Therefore both tools were rejected.

For QPME, we lacked a release of the stable version 2.0, thus we could not use the improvements made in this version. While both LQNS and Palladio workbench offer modeling concepts that are easily mapped onto software modeling concepts, we decided to start modeling using JMT, which feature more intuitive user interfaces and the best documentation. JMT's simplicity in modeling and ability to be directly downloaded contributed to this decision.

Unfortunately, JMT quickly proved not expressive enough. For example, asynchronous behavior could not be directly expressed. Also, the semantic gap between our software design concepts (components interacting by web service calls) and the QNM formalism were an obstacle.

Finally, we opted to use the Palladio workbench, because it supports the simulation of architectural models and because its 'UML-like' interface makes it easier to construct models and communicate them with the stakeholders than LQNS. The ability to re-use models and components was another useful feature [23]. Moreover, the Palladio workbench has been used in industrial case studies before [27, 33], thus we assume that it is mature and sufficiently stable. Palladio's drawbacks lie in its more laborious model creation due to the complex meta model and its weaker user documentation.

4.2 Palladio Component Model

The PCM is based on the component-based software engineering philosophy and distinguishes four developer roles, each modeling part of a system: component developer, software architect, system deployer, and domain expert. Once the model parts from each role have been assembled, the combined model is transformed into a simulation or analysis model to derive the desired performance metrics. Next, we discuss each role's modeling tasks and illustrate the use of the PCM with the ABB RDS model (Fig. 2).

The *component developer role* is responsible for modeling and implementing software components. She puts models of her components and their interfaces in a `Component Repository`, a distinct model container in the PCM. When a developer creates the model of a component she specifies its resource demands for each provided service as well as calls to other components in a so-called `Service Effect Specification` (SEFF).

As an example consider the `SEFF` on the right of Fig. 2. It shows an internal action that requires 100 ms of CPU time and an external call to insert data into the database. Besides mean values, the PCM meta model supports arbitrary

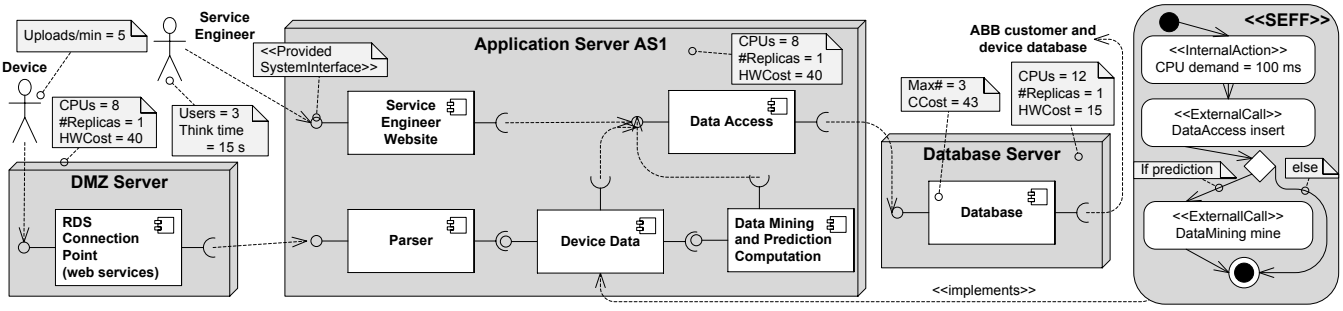

Figure 2: Palladio Component Model of the ABB remote diagnostic system (in UML notation)

probability distributions as resource demands. Other supported resource demands are, for example, the number of threads required from a thread pool or hard disk accesses. The component developer role can also specify component cost (CCost), for example, our database component has cost 10 (CCost = 43) (cf. Section 5.3).

The *software architect role* specifies a `System Model` by selecting components from the component repository and connecting required interfaces to matching provided interfaces. The *system deployer role* specifies a `Resource Environment` containing concrete hardware resources, such as CPU, hard disks, and network connections. These resources can be annotated with hardware costs. For example, application server AS1 in Fig. 2 has 8 CPU cores (CPUs = 8), costs 40 units (HWCost = 40, cf. Section 5.3), and is not replicated (#Replicas = 1).

Finally, the *domain expert role* specifies the usage of the system in a `Usage Model`, which contains scenarios of calls as well as an open or closed workload for each scenario.

The Palladio workbench tool currently provides two performance solvers: SimuCom and PCM2LQN. We chose PCM2LQN in combination with the LQNS analytical solver, because it is usually much faster than SimuCom. Automatic design space exploration requires to evaluate many candidates, so runtime is important. PCM2LQN maps the Palladio models to a layered queueing network (LQN). This does not contradict our former decision against LQNS, since here the LQN models remained transparent to the stakeholders and were only used for model solution. The LQN's analytic solver [26] is restricted compared to SimuCom, since it only provides mean value performance metrics and does not support arbitrary passive resources such as semaphores.

4.3 Model Construction

To model the RDS like in the experimental setup as a Palladio model, we studied its structure and behavior by analyzing the available documentation, talking to its developers, analyzing the source code and performing different load tests. Then, we constructed the model as follows:

Component Repository: Using the existing architectural descriptions to create the component repository formed a major challenge, because these documents were limited in detail and only provided a component level logical view and a deployment view. We used these views to select the components to include in our PCM component repository. Initially, the RDS repository consisted of seven components, seven interfaces, and 27 component services.

SEFFs: To specify the SEFFs for each component service *(step 2)*, we used dynaTrace to analyze the system behavior. We created single stimulus measurements (e.g., a single upload) and analyzed them in depth using the method-level performance information for each transaction in dynaTrace PurePaths [16]. The method level turned out to be far too detailed for a useful behavioral model.

Therefore, we opted to model system behavior at the level of web service calls between the tiers. In some cases, we added more detail to capture differences between use cases. For example, the data mining component needed to be modeled in some detail to get accurate predictions for each type of upload, because the component is quite complex and resource intensive. We further used various overviews of dynaTrace to ensure that we included the most frequent and demanding parts of the software. While we used the aforementioned server log analysis results to limited the SEFFs in our model to represent only the most frequent use cases.

One of the more complex SEFFs is depicted as an example in Fig. 3. While heavily abstracting from the actual source code, it still shows a complex control flow with several resource demands to CPU and hard disk as well as several calls to other components and web services.

While log analyses also revealed that the uploads were not uniformly spread over time, we assumed that the upload rate will be flattened due to easily implementable changes to the system. Since ABB can control the upload schedule to a great extent this is a reasonable assumption. However, we do reserve capacity for (limited) peaks. The reporting functionality of RDS was not included despite its significant performance impact, because there are concrete plans to migrate this functionality to a dedicated server.

Several aspects of the application infrastructure were not considered in the performance model. First of all, the RDS interacts with ABB internal and external 3rd party systems. Our model assumed that the response times of the services offered by these systems do not increase as the load within RDS increases, because we could not easily obtain information about the scalability of these systems.

Second, the Microsoft SQL Server (MS-SQL) database replication/duplication mechanisms were not modeled in detail. The exact behavior of the MS-SQL in these cases is hardly known, and it was not feasible to conduct experiments to prototype the behavior. As a result the database scalability figures and resource requirements are expected to be slightly optimistic.

System Model: From the resulting Palladio component repository, we created a system model instantiating and connecting the components. It contained 7 component instances and 7 connectors. In this case, creating the connections of the components was straightforward.

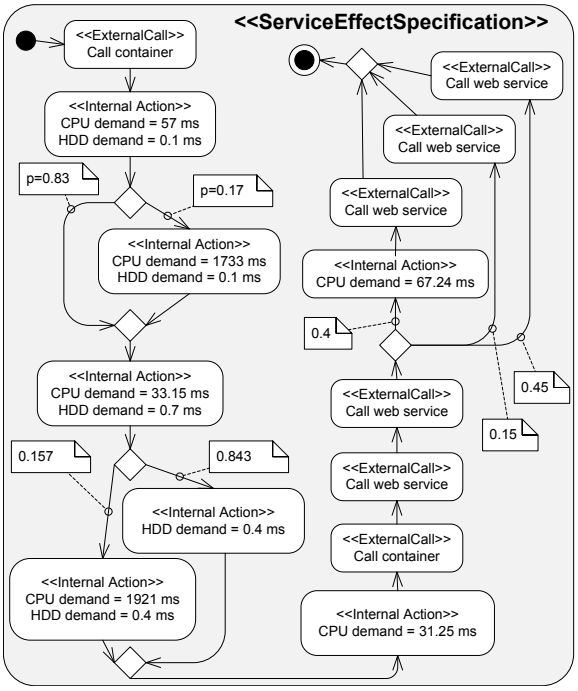

Figure 3: An example service effect specification (SEFF) from the RDS Palladio model showing the inner behavior of one component service in terms of resource demands and calls to other components

Resource Environment: In our case, this model is made up of three servers, each with a CPU and hard disk. The network capacity is assumed to always be sufficient and scaled up by the IT provider as required. The first reason for this assumption is that we expect our IT provider to actually be able to provide the capacity and latency required. The second reason is the limited detail offered by Palladio's network simulator and the subsequent difficulty of specifying the network subsystem in detail. One would have to determine, for a complex system running in .NET, how much latency network messages are issued in each layer.

Allocation Model: We mapped the seven component instances to the three servers in the resource environment according to the allocation in our experimental setup (Fig. 2).

Usage Model: Our usage model reflects the upload service and the service engineering interaction with the system. The former was a periodic request to the system modeled with an open workload and three differently weighted upload types. The latter comprised a more complex user interaction with different branches, loops, and user think times and a closed workload.

4.4 Model Calibration

Calibration of performance models is important to ensure that the resource demands in the model accurately reflect the resource demands in the real system. For calibration of the RDS model, we executed the Palladio performance solvers and compared the predicted utilization for each resource with the utilizations measured by their respective windows performance counters. We conducted this comparison for each of the three workloads introduced in Section 3.3 to assure that the model was robust against different workload intensities.

Despite using the detailed resource demands measured by dynaTrace, the utilizations derived from the initial RDS Palladio model showed a moderate deviation from the actually measured utilizations. Thus, we ran additional experiments and performed code reviews to get a better understanding of the system and why the prediction was off. We focused on those parts of the model where it showed errors of more than 20 % compared to the measurement results. This led to useful insight, either to refine the model or to learn more about the RDS architecture, the system behavior, and bottlenecks in the system. The utilizations derived in each calibration step were recorded in an Excel sheet to track the model accuracy and the effect of changes made to the model.

After calibration the model gives values up to 30% too low for the DMZ server CPU utilization. That means that for a 25% CPU utilization the actual CPU utilization could be 32.5%. The application server utilization figures are off by a maximum of 10% and the database server CPU utilization results are at most 30% too high. Three quarter of the response times for both internal and external calls are within 30% of the measured value.

We report the errors for our high load scenario, because this is most representative of our target workload. The errors for the other two workloads are lower. Overall, the error percentages are reasonable, but not desirably small. However, both our measurements in the experimental setup and our experience during earlier work [23] showed that the application server, for which our model most accurately predicts utilization, would be the most likely bottleneck. There are two main reasons it was not economical to further improve the accuracy of the model. First, the complex behavior of the ASP.Net container especially with our asynchronous application could not be understood within reasonable time. Second, the application behavior was complex, because of its size and the way it was written.

5. DESIGN SPACE EXPLORATION

Based on the created performance model, we want to find cost-efficient architectures to cope with the expected increased workload (cf. Section 3.2). We consider three workload scenarios: Scenario 1 considers a higher workload scenario due to more connected devices. Scenarios 2 and 3 additionally consider an eightfold (scen. 2) and fourfold (scen. 3) increase of processed data per device. For each scenario, we want to determine the architecture that fulfills our main performance goal (50% utilization maximum) at the lowest cost. We first ran several manual predictions using the calibrated model (Section 5.1). Because of the large design space, we applied the automated design space exploration tool 'PerOpteryx' (Section 5.2). As a prerequisite we created a formal PerOpteryx cost model (Section 5.3) and a degree of freedom model (Section 5.4). Finally, we ran several predictions and created an architectural road-map (Section 5.5).

5.1 Manual Exploration

Initially, we partially explored the design space by manually modifying the baseline model [23]. First, we used the AFK scale cube theory, which explains scalability in three fundamental dimensions, the axes of the cube. Scalability can be increased by moving the design along these axes by cloning, performing task-based splits or performing request-based splits. We created three architectural alternatives,

each exploring one axis of the AFK scale cube [19]. Second, we combined several scalability strategies and our knowledge about hardware costs to create further alternatives to cost-effectively meet our capacity goal. Finally, we reflected several updates of the operational software in our model, because the development continued during our study.

The first of our AFK-scale cube inspired alternatives, scales the system by assigning each component to its own server. This complies to the Y-axis in the AFK scale cube. However, some components put higher demands on system resources than others. Therefore, it is inefficient to put each component on its own server. The maximum capacity of this model variant shows that network communication would become a bottleneck.

A move along the X-axis of the AFK scale cube increases replication in a system (e.g., double the number of application servers). All replicas should be identical, which requires database replication. We achieved this by having three databases for two pipelines in the system: one shared-write database and two read-only databases. This scheme is interesting because read-only databases do not have to be updated in real-time.

The AFK scale cube Z-axis also uses replication, but additionally partitions the data. Partitions are based on the data or the sender of a request. For example, processing in the RDS could be split on warning versus diagnostic messages, or the physical location, or owner of the sending device.

All alternatives did not consider operational cost. Therefore, we also developed an informal cost model with hardware cost, software licensing cost and hosting cost. Hardware and hosting costs are provided by ABB's IT provider. A spreadsheet cost model created by the IT provider captures these costs. For software licensing an internal software license price list was integrated with the IT provider's spreadsheet to complete our informal cost model.

We further refined the alternatives with replication after finding a configuration with a balanced utilization of the hardware across all tiers. In the end, we settled on a configuration with one DMZ server running the connection point and parser component, one application server running the other components and one database server only hosting the database, i.e., a 1:1:1 configuration.

To scale up for the expected high workload (scen. 1), we first replicated the application server with an X-split (i.e., two load-balanced application servers, a 1:2:1 configuration). For further workload increase (scen. 2+3), this configuration could be replicated in its entirety for additional capacity (i.e., a 2:4:2 configuration). This resulting architecture should be able to cope with the load, yet it is conservative. For example, no tiers were introduced or removed, and there was no separation based on the request type to different pipelines (i.e., z-split).

The potential design space for the system is prohibitively large and cannot be explored by hand. Thus, both to confirm our results and to find even better solutions, we conducted an automatic exploration of the design space with PerOpteryx, as described in the following.

5.2 PerOpteryx: Automated Exploration

The PerOpteryx tool was designed as an automatic design space exploration tool for PCM models [34, 30]. We selected PerOpteryx because of its ability to explore many degrees of freedom, which sets it apart from similar tools. Additionally, its implementation can directly process PCM models. PerOpteryx applies a meta-heuristic search process on a given PCM model to find new architectural candidates with improved performance or costs. Fig. 4 shows a high-level overview of PerOpteryx's search process:

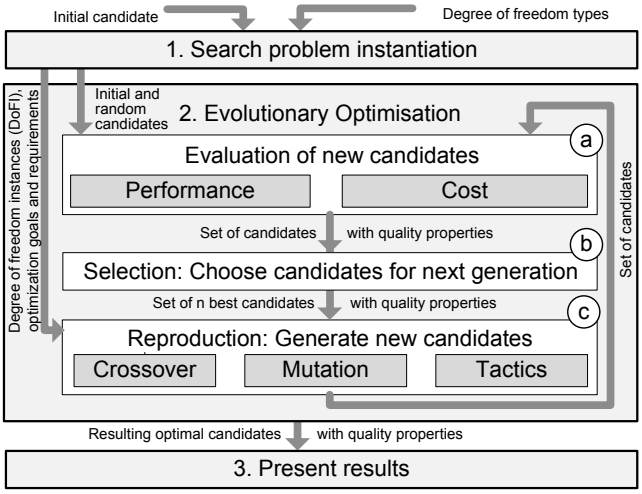

Figure 4: PerOpteryx process model (from [30])

As a prerequisite for applying PerOpteryx, the *degree of freedom types* to consider for optimizing the architecture need to be defined. These types describe how an architecture model may be changed to improve its quality properties [31]. For example, the degree of freedom type "component allocation" describes how components may be re-allocated to different servers that provide the required resources.

In **step 1**, we manually model the search space as a set of *degrees of freedom instances* to explore. Each degree of freedom instance has a set of design options (e.g., a set of CPU clock frequencies between 2 and 4 GHz, or a set of servers a component may be allocated to). Each possible architectural candidate in the search space can be represented relative to the initial PCM model as a set of decisions. This set of decisions—one for each degree instance—is called the genome of the candidate. Furthermore, the *optimization goal and requirements* are modeled in step 1. For example, we can define that the response time of a certain system service and the costs should be optimized, while a given minimum throughput requirement and a given maximum utilization requirement must be fulfilled.

If multiple quality metrics should be optimized, PerOpteryx searches for Pareto-optimal candidates: A candidate is Pareto optimal if there exists no other candidate that is better in all quality metrics. The result of such an optimization 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. If only a single quality metric should be optimized, the minimum or maximum value (depending on the metric) is searched.

In **step 2** PerOpteryx applies evolutionary optimization based on the genomes. This step is fully automated. It uses the NSGA-II algorithm [24], which is one of the advanced elitist multi-objective evolutionary algorithms. 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 (**2a**). The most promising candidates (i.e. close to the current Pareto front, fulfilling the requirements, and well spread) are selected for further manipulation, while the least promising candidates are discarded (**2b**). During reproduction (**2c**), PerOpteryx manipulates the selected candidate genomes using crossover, mutation, or tactics (cf. [30]), and creates a number of new candidates.

From the results (**step 3**), the software architect can identify interesting solutions in the Pareto front fulfilling the user requirements and make well-informed trade-off decisions. To be able to apply PerOpteryx on the RDS Palladio model, we first created a formal PerOpteryx cost model (Section 5.3) and a degree of freedom instances model (Section 5.4) as described in the following two subsections.

5.3 Formal RDS Cost Model

The PerOpteryx cost model allows to annotate both hardware resources and software components with the total cost of ownership, so that the overall costs can be derived by summing up all annotations. For our case study, we model the total costs for a multiple year period, which is reasonable since the hosting contract has a minimum duration of several years. In total our cost model contained 7 hardware resource and 6 software component cost annotations. The hardware resource costs were a function depending on the number of cores used.

However, the cost prediction cannot be fully accurate. First, prices are re-negotiated every year. Second, we can only coarsely approximate the future disk storage demands. Finally, we do not have access to price information for strategic global hosting options, which means that we cannot explore the viability of replicating the RDS in various geographical locations to lower cost and latency.

Furthermore, we are unable to express between different types of leases. The IT provider offers both physical and virtual machines for lease to ABB. The two main differences are that virtual machines have a much shorter minimum lease duration and that the price for the same computational power will drop more significantly over time than for physical servers. While these aspects are not of major impact on what is the best trade-off between price and performance, it has to be kept in mind that a longer lease for physical machines that have constant capacity and price (whereas virtual machines will become cheaper for constant capacity) reduces flexibility and may hurdle future expansion.

5.4 Degrees of Freedom and Goal

For the ABB RDS system, we identified and modeled three relevant degree of freedom types:

Component allocation may be altered by shifting components from one resources container to another. However, there are restriction to not deploy all components on the DMZ servers and to deploy database components on specific configurations recommended by the IT provider. With four additional resource containers as potential application servers in the model, PerOpteryx can explore a Y-axis split with one dedicated application server per component.

Resource container replication clones a resource container including all contained components. In our model, all resource containers may be replicated. We defined the upper limits for replication based on the experience from our manual exploration [23]. If database components are replicated, an additional overhead occurs between them to communicate their state. This is supported by our degree of freedom concept, as it allows to change multiple elements of the architecture model together [31]. Thus, we reflected this synchronization overhead by modeling different database component versions, one for each replication level.

Number of (CPU) cores can be varied to increase or decrease the capacity of a resource container. To support this degree of freedom type, the cost model describes hardware cost of a resource container relative to the number of cores. The resulting design space has 18 degree of freedom instances:

- 5 *component allocation* choices for the five components initially allocated to application server AS1: They may be allocated to any of the five application servers and to either the DMZ server or the database server, depending on security and compatibility considerations.
- 6 *number of (CPU) cores* choices for the five available application servers and the DMZ server, each instance allows to use 1, 2, 3, 4, 6, or 8 cores as offered by our IT provider.
- 7 *resource container replication* choices for the five application servers (1 to 8 replicas), the DMZ server (1 to 8 replicas), and the database server (1 to 3 replicas). The *resource container replication* degree of freedom instance for the database server also changes the used version of the database component to reflect the synchronization overhead of different replication levels.

The size of this design space is the combination of choices within these instances and thus is 3.67×10^{15} possible architecture candidates.

The degree of freedom types "Resource container replication" and "Number of cores" have been newly defined for this work. As such, the combined optimization of software architectures along all three degree of freedom types has not been presented before and shows the extensibility of PerOpteryx. Furthermore, the possibility to model the changing database behavior (due to synchronization) for different number of replicas shows the flexibility of PerOpteryx' degree of freedom concept.

The goal of the design space exploration for a given workload is to find an architectural candidate that minimizes costs while fulfilling performance requirements. Three performance requirements are relevant: First, the response time of service engineers when calling a system service should be below a given threshold. Second, the response time of the upload service called by the devices should be below a given threshold to ensure the timeliness of the data. Finally, the CPU utilization of all used servers should be below 50%.

5.5 Automated Exploration Results

In the following, we present the exploration results for the three scenarios. As I/O activity is not modeled in the RDS performance model, PerOpteryx cannot take it into account. This assumption has to be validated in future work.

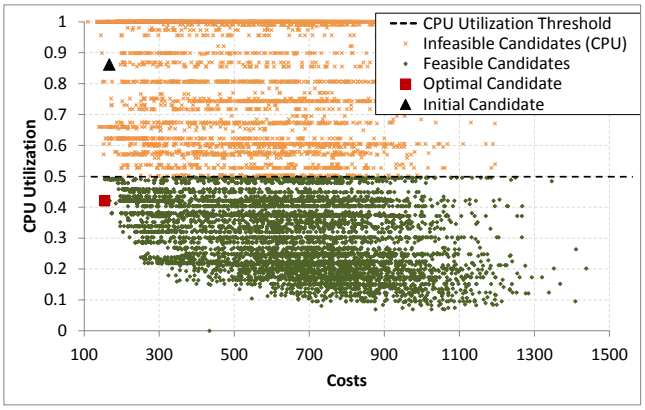

Figure 5: Evaluated Architecture Candidates for High Workload Scenario (Scenario 1). The line at 50% CPU utilization separates feasible candidates (below) from infeasible ones (above).

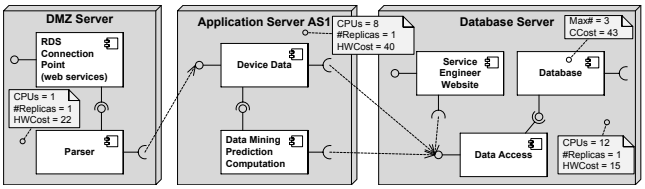

Figure 6: Found Optimal Architecture for High Workload (Scenario 1)

5.5.1 *Scenario 1: Higher Workload*

For the higher workload scenario, we first ran 3 PerOpteryx explorations of the full design space in parallel on a quad-core machine. Each run took approx. 8 hours. Analyzing the results, we found that the system does not need many servers to cope with the load. Thus, to refine the results, we reduced the design space to use only up to three application servers, and ran another 3 PerOpteryx explorations. Altogether, 17,857 architectural candidates were evaluated.

Fig. 5 shows all architecture candidates evaluated during the design space exploration. They are plotted for their costs and the maximum CPU utilization, which is the highest utilized server among all used servers.

Candidates marked with a cross (×) have a too high CPU utilization (above the threshold of 50%). Overloaded candidates are plotted as having a CPU utilization of 1. The response time requirements are fulfilled by all architecture candidates that fulfill the utilization requirement.

Many candidates fulfilling all requirements have been found, with varying costs. The optimal candidate (i.e. the candidate with the lowest costs) is marked by a square. This optimal candidate uses three servers (DMZ server, DB server, and one application server) and distributes the components to them as shown in Fig. 6. Some components are moved to the DMZ and DB server, compared to the initial candidate. No replication has to be introduced, which would lead to unnecessarily high costs. Furthermore, the number of cores of the DMZ server are reduced in the optimal candidate to save additional costs.

Note, that we did not consider the potentially increased reliability of the system due to replication. A reliability model could be added to reflect this, so that PerOpteryx could also explore this quality dimension (as for example done in [34]).

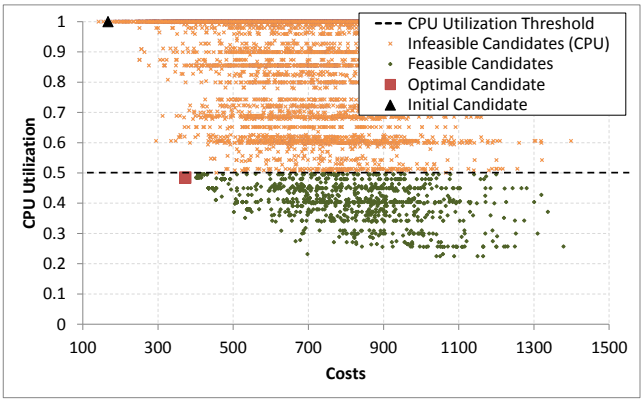

Figure 7: Evaluated Architecture Candidates for High Workload and Information Growth 8 (Scenario 2)

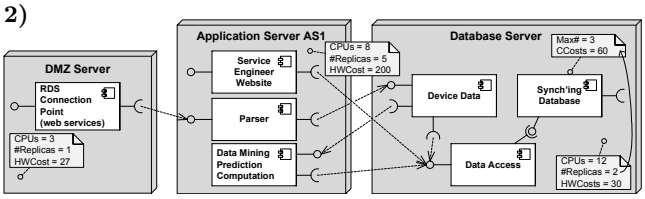

Figure 8: Found Optimal Architecture for High Workload and Information Growth 8 (Scenario 2)

5.5.2 *Scenario 2: Higher Workload and Information Growth*

If each device sends more data for processing, this leads to an increased demand of some of the components per device request. Thus, the overall load of the system increases further. In this scenario 2, we assume an increase of device information by a factor 8, which leads to higher resource demands in some components where the computation is dependent on the amount of processed data. The new demands were modeled by adding a scalar to the original demands. We defined the scalars based on the theoretical complexity of the operation. For example, a database write scales linearly with the amount of data to be written.

8436 candidates have been evaluated for this scenario in 3 parallel PerOpteryx runs, each running for approx. 8 hours. Fig. 7 shows the evaluated candidates. Compared to the previous scenario, fewer candidates have been evaluated because only the full design space has been explored. More of the evaluated candidates are infeasible or even overloaded and the feasible candidates have higher costs, as expected for the increased workload. The initial candidate as shown in Fig. 2 and the optimal candidate found for the previous scenario 1 are overloaded in this workload situation.

The found optimal candidate is shown in Fig. 8. The components are allocated differently to the servers. Additionally, five replicas of the application server and 2 replicas of the database server are used. This also leads to higher component costs for the database, as two instances have to be paid for. Still PerOpteryx found it beneficial to use the database server as well and even add components to it, because the (physical) database server is less expensive relative to computing power (recall Section 5.3).

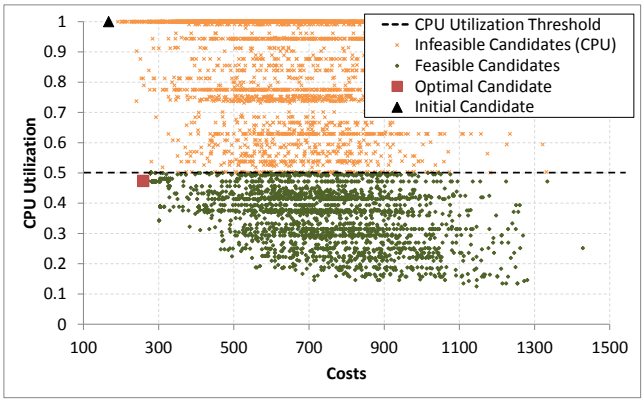

Figure 9: Evaluated Architecture Candidates for High Workload and Information Growth 4 (Scenario 3)

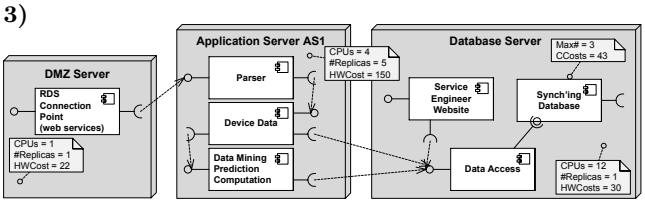

Figure 10: Found Optimal Architecture for High Workload and Information Growth 4 (Scenario 3)

5.5.3 *Scenario 3: Higher Workload and Intermediate Information Growth*

As a migration step from scenario 1 to scenario 2 with information growth, we additionally analyzed an intermediate information growth of a factor 4. The PerOpteryx setup and run statistics are comparable to scenario 2. Fig. 9 shows the evaluated candidates. As expected, the cloud of evaluated candidates lies in between the results of scenario 1 and 2. For example, there are fewer feasible candidates than in scenario 1, but more than in scenario 2.

Fig. 10 shows the resulting optimal candidate. Compared to the optimal candidate from scenario 1 (Fig. 6), PerOpteryx has moved the Parser component to the application server as well, to be able to use only one DMZ server. The database server is unchanged. The application server has been strengthened to cope with the increased load and the additional Parser component.

However, additional manual exploration shows that the candidate is not truly optimal: PerOpteryx chose to use 5 replicas with 4 cores each here. After inspecting PerOpteryx' optimal candidates, we found that an application server with 3 replicas and 8 cores each would actually be even slightly faster and cheaper (only costs of 120 instead of 150). Thus, a longer exploration run would be required here for a truly optimal solution. Alternatively, we could devise additional PerOpteryx tactics that first analyze the costs for replication of cores and servers and then adjust the model to achieve the cheapest configuration with equivalent processing power. Note, however, that PerOpteryx' automation still is beneficial, as it would be laborious or even impossible to come to these conclusions with manual exploration only.

5.5.4 *Summary and Recommendations*

Based on these results, we can recommend a road-map for scaling the RDS. First, to cope with the expected workload increase (scenario 1), the system should be configured in a three tier configuration as shown in Fig. 6. During our manual exploration we made a similar conclusion with regards to the DMZ server. However, we did not know which components to off-load from the application server to the database server. We did consider to place both the data access and data mining predictions on the database server but this overloaded the database server. Hence, the optimal solution for scenario 1 is a partial surprise, but is still valid.

If the workload becomes higher (e.g. due to information growth, scenario 3), the application server should host more components and should be replicated as shown in Fig. 10. Finally, a further increased workload due to more information growth (scenario 2) requires to replicate all three tiers as shown in Fig. 8, while at the same time the allocation of components to application server and database server is slightly adjusted to make optimal use of the processing power. Based one these findings, we formulated a 5 year road-map for the future development of the system. We plan to validate our evaluation after 2 years, as the first steps in the road-map have been realized.

6. LESSONS LEARNED

In this section we share the lessons that we took from our study and that we consider of value to other industry practitioners. Researchers may find ideas on how to improve their performance modeling techniques to meet industry needs.

Performance modeling increases understanding.

The performance modeling proved useful in itself, because it forced us to understand the system's (performance) behavior and identify the bottlenecks. It helped us to ask the right questions about the system and gave us insight that was potentially just as valuable as capacity predictions. For example, model calibration helped us to find oddities in the system behavior. The model represents a polished version of the system that should match its average behavior, but under varying loads measurements and predictions occasionally diverge. One of the things we learned during calibration was that a lock statement was put in the code to limit the amount of concurrently running data mining processes, as to free resources for the internal website that was running on the same server.

Predictions shift stakeholder discussion.

The discussion about the architectural road-map with our stakeholders changed once we introduced the model predictions. The data shifted the conversation from discussion towards a situation where we explained the modeling/evaluation results and the road-map was more or less taken for granted. There was no longer discussion about what the way forward should be. This means that the credibility of our study was high, despite or maybe due to the fact that we presented our stakeholders with a detailed overview of the assumptions underlying the model, their effect on accuracy, and a list of things we did to ensure accuracy.

Economic benefit must exist.

The cost of measuring and modeling are quite high. One has to consider the cost for load generator and measurement tools, training, an experimental copy of the system, and human resources. The latter include the strain on developers and product owners, in addition to the cost for the performance study team. Our study took approximately four full-time employees six months. Adding everything up, one can

conclude this type of projects are too expensive for small systems. Short-term fixes may turn-out to be cheaper, despite their inefficiency. We are therefore not surprised that performance fire-fighting is still common practice. More support for performance modelers would be required to decrease the needed effort, e.g. by automatically creating initial performance models based on log data.

Corporate processes may stall license agreements.

It is important to take into account the time required to reach license agreements with vendors. We encountered two problems. First, the license model of software vendors may not fit multi-national companies that have the need to migrate their licenses between machines in different countries. Second, academic software owners do not realize how tedious corporate processes are and how even their simple license hurdles corporate use of their software. For example, due to the need for non-standard licenses to be reviewed by legal experts. This is unfortunate because corporations can often afford to be early adaptors of new technology due to the expertise and money they have available to experiment.

Performance of performance modeling tools limited.

Even for modestly sized systems such as the RDS the performance of the performance modeling tools may be a problem. In our earlier study, we could not use the standard distribution of Palladio workbench, because it ran out of memory [23]. In this study, we reverted to the LQNS to limit the runtime of our design space exploration. The scalability of the modeling formalism also proved to be important. We could comfortably model the RDS and various architectural variations, but we think that the model complexity will be significant for systems that are two times bigger.

It pays off to invest in good tools.

It is difficult to overemphasize the convenience of having the right tools. The combination of dynaTrace and NeoLoad enabled us to take an enormous amount of measurements, and to navigate these easily. In practice, this meant that we could easily study the effect of different software and hardware configurations on performance. The changing of hardware configurations was enabled by using virtual machines in our experimental setup. The repository of performance measurements, which included over 100 load test runs, was frequently consulted during model construction.

7. RELATED WORK

Our work uses the foundations of software performance engineering [39, 21, 32] and multi-objective meta-heuristic optimization [22]. We compare our approach to (i) recent industrial case studies on performance prediction and (ii) recent design space exploration approaches in the software performance domain.

Most recent industrial case studies on performance modeling are restricted to a limited number of evaluated design alternatives. Liu and Gorton [36] constructed a queueing network for an EJB application and conducted a capacity planning study, predicting the throughput for a growing number of database connections. Kounev [29] built a queuing Petri net for the specJAppServer2004. The author measured the system for different workloads and analyzed the impact of a higher number of application server nodes (i.e., 2,4,6,8) for various performance metrics.

Jin et al. [28] modeled the performance of a meter-data system for utilities with a layered queuing network model. They constructed a very large LQN with more than 20 processors and over 100 tasks. After benchmarking the system, they analyzed the throughput of the system for massively higher workloads. Huber et al. [27] built a Palladio Component Model instance for a storage virtualization system from IBM. They measured performance of the system and analyzed the performance for a synchronous and asynchronous re-design using the model. All of the listed case studies analyze only a single degree of freedom and/or changing workload and do not explicitly address the performance and costs trade-offs for different alternatives.

Concerning design space exploration approaches in the performance domain, three main classes can be distinguished: *Rule-based approaches* improve the architecture by applying predefined actions under certain conditions. *Specialized optimization approaches* have been suggested for certain problem formulations, but they are limited to one or few degree of freedom types at a time. *Meta-heuristic approaches* apply general, often stochastic search strategies to improve the architecture. They use limited knowledge about the search problem itself.

Two recent rule-based approaches are PerformanceBooster and ArchE. With PerformanceBooster, Xu et al. [40] 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. [25] 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. Rule-based approaches share the two limitations of being restricted to the pre-defined improvement rules and the potential of getting stuck in local optima.

Two recent meta-heuristic approaches are ArcheOpteryx and SASSY. Aleti et al.[20] use ArcheOpteryx 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. Menascé et al. [37] generate service-oriented architectures using SASSY that satisfy quality requirements, using service selection and architectural patterns. They use random-restart hill-climbing. All meta-heuristic-based approaches to software architecture improvement explore only one or few degrees of freedom of the architectural model. The combination of component allocation, replication, and hardware changes as supported by PerOpteryx is not supported by the other approaches, and furthermore PerOpteryx is extensible by plugging in additional model transformations [31].

In addition, the other approaches target to mitigate existing performance problems or improve quality properties, while our study targets to optimize costs while maintaining acceptable performance (still, other quality properties can also be optimized with our approach, if reasonable in setting at hand).

8. CONCLUSIONS

This paper has demonstrated how to construct a component-based performance model using state of the art tools for measuring and modeling. We applied the automatic design space exploration tool PerOpteryx on this model and

evaluated more than 33,000 architectural candidates for an optimal trade-off between performance and costs in three scenarios. Our case study resulted in a migration road-map for a cost-effective evolution of the existing system.

Our approach enables ABB to comply with future performance requirements (thus helping in sales) and to avoid poor architectural solutions (thus improving development efficiency). It also helps in better understanding the performance impacts on the system and is thus instrumental in performance tuning. Other practitioners can draw from our experiences. For researchers, we have demonstrated that automated design space exploration is feasible for a complex industrial system albeit incurring significant costs. We have created a detailed performance models and found pointers for future research.

Performance measurement and modeling should become more tightly integrated (e.g., by creating Palladio models automatically from dynaTrace results). Network modeling was rather abstract in our study due to the lack of support in the Palladio model. More detailed modeling of the network could lead to even more accurate prediction results. There is potential to automatically draw feasible migration road-maps from the PerOpteryx results for different workloads by highlighting compatible candidates. This should be investigated in future research in more detail.

9. REFERENCES

[1] ArgoSPE plug-in for ArgoUML. argospe.tigris.org/.

[2] Core Scenario Model. www.sce.carleton.ca/rads/puma/.

[3] Java Modelling Tools. jmt.sourceforge.net/.

[4] KlaperSuite. klaper.sourceforge.net/.

[5] Layered Queueing Network Solver software package. www.sce.carleton.ca/rads/lqns/.

[6] Möbius tool. www.mobius.illinois.edu/.

[7] Oris Tool. www.stlab.dsi.unifi.it/oris/.

[8] Palladio Software Architecture Simulator. www.palladio-simulator.com/.

[9] PEPA Tools. www.dcs.ed.ac.uk/pepa/tools/.

[10] PRISM probabilistic model checker. www.prismmodelchecker.org/.

[11] QPME – Queueing Petri net Modeling Environment. descartes.ipd.kit.edu/projects/qpme/.

[12] SHARPE. people.ee.duke.edu/~kst/software_packages.html.

[13] SPE-ED Performance Modeling Tool. www.perfeng.com/sped.htm.

[14] TwoTowers tool. www.sti.uniurb.it/bernardo/twotowers/.

[15] GreatSPN – GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. www.di.unito.it/~greatspn, 2008.

[16] Dynatrace – Application Performance Management and Monitoring. www.dynatrace.com, 2011.

[17] Neotys Neoload Load Testing Tool. www.neotys.com/product/overview-neoload.html, 2011.

[18] Sawmill – Universal log file analysis tool. www.sawmill.net, 2011.

[19] M. L. Abbott and M. T. Fisher. *The art of scalability.* Addison–Wesley, 2009.

[20] 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.

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

[22] 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.

[23] T. de Gooijer. Performance Modeling of ASP.Net Web Service Applications: an Industrial Case Study. Master's thesis, Mälardalen University, Västerås, Sweden, 2011.

[24] 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.

[25] A. Díaz Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating quality-attribute reasoning frameworks in the archE design assistant. In *Proc. 4th Int. Conf. on the Quality of Software-Architectures (QoSA 2008)*, volume 5281, pages 171–188, 2008.

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

[27] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. Reussner. Performance modeling in industry: a case study on storage virtualization. In *Proc. of ICSE'10*, pages 1–10. ACM, 2010.

[28] Y. Jin, A. Tang, J. Han, and Y. Liu. Performance Evaluation and Prediction for Legacy Information Systems. In *Proc. of ICSE'07*, pages 540–549. Ieee, May 2007.

[29] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. on SE*, 32(7):486–502, July 2006.

[30] A. Koziolek, H. Koziolek, and R. Reussner. PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization. In *Proc. 7th Int. Conf. on the Quality of Software Architectures (QoSA'11)*, pages 33–42. ACM, 2011.

[31] A. Koziolek and R. Reussner. Towards a generic quality optimisation framework for component-based system models. In *Proc. 14th Int. ACM Sigsoft Symposium on Component-based Software Engineering (CBSE'11)*, pages 103–108. ACM, June 2011.

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

[33] H. Koziolek, B. Schlich, C. Bilich, R. Weiss, S. Becker, K. Krogmann, M. Trifu, R. Mirandola, and A. Koziolek. An Industrial Case Study on Quality Impact Prediction for Evolving Service-Oriented Software. In *Proc. of ICSE'11, SEIP Track*. ACM, May 2011.

[34] A. Koziolek (Martens), H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. *Proceedings of ICPE'10*, pages 105–116, January 2010.

[35] C. Letner and R. Gimarc. A Methodology for Predicting the Scalability of Distributed Production Systems. In *CMG Conference*, volume 1, page 223. Computer Measurement Group; 1997, 2005.

[36] V. Liu, I. Gorton, and A. Fekete. Design-level performance prediction of component-based applications. *IEEE Trans. on SE*, 31(11):928–941, Nov. 2005.

[37] 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 *Proceedings of ICPE'10*, pages 27–36. ACM, 2010.

[38] S. T. Redwine JR and W. E. Riddle. Software Technology Maturation. In *Proc. of ICSE'85*, pages 189–200. IEEE, 1985.

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

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