Multi-Objective Optimization of Deployment Topologies for Distributed Applications

FELIX WILLNECKER, fortiss GmbH
HELMUT KRCMAR, Technical University of Munich (TUM)

Modern applications are typically implemented as distributed systems comprising several components. Deciding where to deploy which component is a difficult task that today is usually assisted by logical topology recommendations. Choosing inefficient topologies allocates the wrong amount of resources, leads to unnecessary operation costs, or results in poor performance. Testing different topologies to find good solutions takes a lot of time and might delay productive operations. Therefore, this work introduces a software-based deployment topology optimization approach for distributed applications. We use an enhanced performance model generator that extracts models from operational monitoring data of running applications. The extracted model is used to simulate performance metrics (e.g., resource utilization, response times, throughput) and runtime costs of distributed applications. Subsequently, we introduce a deployment topology optimizer, which selects an optimized topology for a specified workload and considers on-premise, cloud, and hybrid topologies. The following three optimization goals are presented in this work: (i) minimum response time for an optimized user experience, (ii) approximate resource utilization around certain peaks, and (iii) minimum cost for running the application. To evaluate the approach, we use the SPECjEnterpriseNEXT industry benchmark as distributed application in an on-premise and in a cloud/on-premise hybrid environment. The evaluation demonstrates the accuracy of the simulation compared to the actual deployment by deploying an optimized topology and comparing measurements with simulation results.

CCS Concepts:
• Software and its engineering → Software evolution; Software configuration management and version control systems; Software maintenance tools; Maintaining software; Software as a service orchestration system;

Additional Key Words and Phrases: Deployment topology optimization, performance model, distributed enterprise applications, performance model generation, memory simulation

ACM Reference Format:

1. INTRODUCTION

Distributed architectures are state of the art in large-scale and modern internet applications [Brunnert and Krcmar 2015; Brunnert et al. 2015]. Such applications typically comprise multiple deployment units. Each deployment unit is composed of several components and is movable from one server instance to another. Furthermore, these units can be replicated to cope with increased workload based on data from operations. Selecting the right amount of deployment unit replications and corresponding runtime instances is a difficult task and requires developers (dev) and operations (ops) expertise. Numerous different combinations, so-called deployment topologies, exist [Brunn...
Not only the right amount of unit replications must be selected, but also the right amount of resource containers (e.g., Virtual Machines (VMs), bare-metal server, or application containers), which depends on the demand a component operation places on its resources [Brunnert et al. 2015]. The most important resources, including Central Processing Unit (CPU), Hard Disk Drive (HDD), memory and network, and their demands have to be considered for deployment topology decisions [Koziolek et al. 2014; Brunnert et al. 2015].

At present, these decisions are assisted by logical recommendations or simply based on estimations instead of continuous measurements from operations [Brunnert et al. 2015]. Such estimations and resulting topologies often rely on peak demands, which leads to under-utilized data centers [Speitkamp and Bichler 2010]. Different studies estimate the current average CPU utilization in data centers between 6% and 20% [Huang and Masanet 2015; Speitkamp and Bichler 2010]. Hence, servers are idle for most of their uptime. This situation serves well for Infrastructure as a Service (IaaS) cloud providers as they can over-provision their physical capacities, but the situation produces unnecessary costs for operators of distributed applications (DAs). By contrast, over-utilized servers are not desirable as this results in overly long response times or unstable systems. Therefore, selecting the right amount of resource containers and optimizing the utilization of their resources is important when running DAs [Ardagna et al. 2014].

Instead of deploying a DA on-premise, managed infrastructures like cloud environments are available today and can provide extensive reliability and cost reduction [Ardagna et al. 2014]. Managed infrastructure providers now invoice the usage of runtime instances. In contrast to previously purchased servers, these providers charge current costs. Thus, DA operators have a vested interest in optimizing their topologies in order to reduce operation costs. Different providers apply different cost models based on the number of machines, requests processed, number of user sessions, or simply aggregated uptime. Depending on which provider is chosen, the optimization goal for the deployment might change to save costs. A high utilization might be preferred if the uptime of servers forms the basis of the cost model instead of optimized response times.

In practice, deployment topology considerations require a great amount of effort and expert knowledge about operation data and the software that has been developed. The role of the DevOps engineer is designed for these type of tasks. Planning and testing topology changes in a production environment may incur risks for the stability of the DA. The potential savings in operation costs or performance gains compared to the risks might not be worth it. Evaluating topologies in test environments requires production-like environments, although these environments are as expensive as the production environment itself. Furthermore, such testing environments are often used to capacity by various projects executing load tests or may simply not yet be available when new hardware or managed infrastructures like cloud environments are introduced as new target environments [Ardagna et al. 2014].

This work proposes to use performance models extracted from small scale test environments and subsequently size and optimize available resource containers to specified workloads. To accomplish this goal, we combine performance model generators (PMGs) and architecture optimizers to automatically detect optimized deployment topologies. This approach allows to use accurate resource demands from generated performance models and established architecture optimization algorithms. We use the PMG of the RETIT1 Capacity Manager, the Palladio Component Model (PCM) as performance meta-model, and an opt4j based approach to optimize the deployment.

1http://www.retit.de/
topologies for DAs [Brunnert and Krcmar 2015; Koziolek et al. 2011; Lukasiewycz et al. 2011].

Performance models can be used to predict performance metrics by evaluating alternative deployment topologies and resource environments, and simulate the effects on these metrics [Brunnert and Krcmar 2015]. Building such models manually often outweighs any potential benefit [Kounev 2005]. Recent research created PMGs for DAs to limit the effort of building such models [Brunnert and Krcmar 2015]. However, no currently available PMG considers all four main resources (CPU, HDD, memory, and network). The PMG we use has a comprehensive approach, but lacks automatic memory management simulation. Therefore, we introduce an extension to PCM and the contemplated PMG for dynamic and automatic memory management modeling and simulation.

PMGs focus on the extraction of the software architecture of a DA but disregard deployment topology decisions. Selecting the right amount of resources and evaluating the selected topologies again requires manual effort. Manual selection soon becomes impossible as the number of potential topologies grows exponentially with the number of deployment units and available resource containers [Koziolek et al. 2011]. To automate this process architecture optimizers have been introduced to the scientific community [Aleti et al. 2013; Koziolek et al. 2011]. These optimizers require an already created (performance) model to conduct optimizations [Koziolek et al. 2011]. Such models can be derived from design specifications or created manually. However, the actual resource demands are usually estimated and therefore error-prone. While generated performance models provide high accuracy for predictions, they are not yet compatible with architecture optimizers. Development and evaluation of an automated approach based on PMG and architecture optimizers to identify ideal deployment topology is the main contribution of this work.

Our results allow DevOps engineers to evaluate different resource environments (e.g., in-house, hosted, cloud), to evaluate different deployment topologies, and to automatically size DAs without deploying the application in a production or production-like environment. For the evaluation we conducted a series of controlled experiments using the industry standard benchmark SPECjEnterpriseNEXT as DA.

To what is already known in this area we contribute:

(i) An automated approach to identify optimal deployments.
(ii) The combination of performance model generation and architecture optimization.
(iii) System design and evaluation of automatic deployment topology selection in different resource environments.
(iv) A dynamic and automatic memory management simulation approach.
(v) A cost model for on-premise and IaaS cloud environments.

This paper builds on our previous work [Willnecker and Krcmar 2016; Willnecker et al. 2015] on deployment topology optimization and contains the following major improvements and extensions:

(i) Multi-objective optimization that calculates the Pareto-front along the optimal results for three goals: minimum response time, minimum costs, optimal resource utilization.
(ii) A flexible cost-model for cloud and on-premise environments based on actual usage of the resources.
(iii) An extension for our garbage collection approach that allows growing and shrinking committed memory.

SPECjEnterpriseNEXT is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterpriseNEXT results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result.
(iv) An evaluation of a newer and more complex SPECjEnterpriseNEXT version in an on-premise and industry cloud environment.

2. RELATED WORK

Early PMGs have been demonstrated in the work of Hrischuk et al. in 1999 [Hrischuk et al. 1999]. Their work focuses on layered queueing networks (LQNs) which do not separate workload, software components and resource environments [Hrischuk et al. 1999]. Without such a separation, exchanging the resource environment model or changing the deployment topology is difficult to accomplish. Therefore, architecture-level performance models, such as PCM, introduce separated sub-models for workload, software architecture and resource environments [Becker et al. 2009]. The work of Brosig et al. (2014) generates performance models for PCM and simulates CPU, HDD and memory demands, but lacks automatic memory management and network demands [Brosig et al. 2014]. Especially in distributed environments the network latency and bandwidth can have a huge impact on the performance of the system [Brunnert and Krcmar 2015].

Another performance model generation approach has been introduced by Brunnert et al. (2015) [Brunnert and Krcmar 2015]. The generated models are called resource profiles and consider CPU, HDD, and network demands [Brunnert and Krcmar 2015]. The approach generates accurate models from running DAs but lacks automatic memory management [Brunnert and Krcmar 2015]. Thoroughly conducted capacity planning requires taking the memory resource into account in order to charge the capacity of available systems effectively. We extend this generator with automatic memory management simulations in our deployment topology optimization architecture.

Huber et al. (2016) describe a performance model-based approach for self-aware systems [Huber et al. 2016]. An endless monitoring, model deduction, and prediction loop allows the constant optimization of the performance of an application system [Huber et al. 2016]. The effects rely on the quality of the workload prediction. If the correct workload for a certain timeframe is predicted, the topology can be adjusted. Nevertheless, complex optimizations are often prohibited as decisions must be made fast.

Speitkamp et al. (2010) identified the need to consolidate resource usage and proposed a mathematical model to optimize resource allocation using VMs [Speitkamp and Bichler 2010]. VMs with high CPU utilization could be run on the same host together with VMs utilizing other resources having low CPU utilization [Speitkamp and Bichler 2010]. This concept should optimize the utilization of all resources in a data center. However, the proposed model is not aware of the workload or the DAs running in the VMs and their dependencies. The model requires a re-calculation and allocation of the VMs when the resource utilization of the hosted applications changes. Such changes occur frequently as new versions are deployed or the executed workload changes.

Chen et al. (2015) demonstrate a tool called StressCloud to model and optimize cloud deployments. The focus on performance and energy consumption as well as load test generation [Chen et al. 2015]. This approach allows to model workload, software and target infrastructure in a new meta-model [Chen et al. 2015]. The accuracy of such approaches relies on the quality of the model. A model generation approach based on measurement results may increase the quality of the StressCloud models.

The architecture optimization approach PerOpteryx evaluates design alternatives based on PCM models [Koziolek et al. 2011]. The number of decisions is large as hardware, network, and software architecture are taken into account. PerOpteryx has a broad variety of optimization goals and degrees of freedom. We adapt the PerOpteryx approach for deployment optimization but with certain changes (e.g., simulations instead of analytical solvers and limitation of deployment topology decisions). This adap-
3. DISTRIBUTED APPLICATION COMPONENTS

A deployment unit is a packaged artifact installable on a server instance or directly on an operating system (OS). Such units consist of several components and operations and build the core of any DA. A typical DA consists of many deployment units distributed throughout multiple servers.

In order to analyze the deployment topology of a DA, its context has to be taken into account. The context comprises (i) the resource profile of the DA and (ii) the workload executed on the DA. Hence, an optimized topology always depends on both factors.

Figure 1 depicts the main components of a DA. We use Java Enterprise Edition (EE)
as an example middleware even though the depicted concepts are applicable for other technologies as long as monitoring technology is available [Spinner et al. 2015; Willnecker and Krcmar 2016]. Figure 1 also shows how we map the different parts of a DA to the PCM meta-model. We selected PCM as it is a mature and stable meta-model and corresponding simulation environment. Our previous technology, like the PMG, extensions to simulate HDDs, and network have already been applied to PCM and we can easily reuse these accomplishments [Becker et al. 2009; Brunnert and Krcmar 2015].

The workload describes the number of users and how they use the DA, which ultimately causes the resource utilization of the DA. These users can be real users accessing the system or virtual users executing a (load) test to analyze the behavior of a DA. We use Application Performance Management (APM) data during a monitoring run to derive an initial workload. This workload is editable in order to size a DA according to the expected workload in production.

The resource profile of a DA describes how an operation of a component utilizes different resources [Brunnert and Krcmar 2015]. The profile consists of a basic workflow and the deployment unit structure of the DA. Each operation of the DA is modeled in the profile including its resource demands for several resources. Resource profiles are the core result of PMGs [Brunnert and Krcmar 2015].

The runtime of the DA representation is defined by the deployment topology. For each deployment unit of a DA at least one instance exists during runtime. If required by the workload, replicas of a deployment unit might exist. If depicted as a graph, each deployment unit node needs at least one edge to a resource container node. The node is replicated if multiple edges from a deployment unit node exist.

A deployment topology describes the structure and relationship of a set of these deployment units installed and executed on a number of resource containers. The containers are organized in the so-called resource environment model. This environment consists of the hosting machines, their capabilities (e.g., CPU processing rate, available memory, HDD speed), and the network connections (focusing on bandwidth and latency) between the resource containers. Two deployment units, which are dependent, can only be deployed on two resource containers that are linked via a network connection. Therefore, the number of potential topologies depends on the number of deployment units $du$ and the valid resource containers $rc$. Equation 1 calculates the number of possible deployment topologies depending on $du$ and $rc$, when all resource container targets are valid for all deployment units.

$$ DT_{du,rc} = (2^{rc} - 1)^{du} $$

$2^{rc} - 1$ describes each possible installation combination of a deployment unit on one or more resource containers. As any permutation with other deployment unit installations is possible, we have to add $du$ as an exponent. Given 10 resource containers and 5 deployment units, the number of possible topologies is already greater than $10^{15}$. The number of combinations in this scenario prohibits a manual selection. Deployment topology optimization requires an automated approach.

4. DEPLOYMENT TOPOLOGY OPTIMIZATION PROCESS

An automated approach requires a holistic tool to optimize deployment topologies, which comprises three basic components:

(i) Performance model generator to detect the resource profile of a DA including its resource demands, system behavior, current deployment topology, and current workload.

(ii) Architecture optimizer to evaluate different target deployment topologies and to select the best topology in terms of the optimization goal.
(iii) Simulation service for parallel predicting performance metrics of multiple performance models.

Figure 2 illustrates the optimization process. We deploy a DA in a test environment to conduct the process. This DA is instrumented with APM agents and set under load in order to obtain meaningful APM data. In a first step, this APM data is used to generate a performance model. The model consists of the detected workload, the detected resource profile, and a specification of the resources in the test environment. The generated model represents the current state of the DA in the test environment.

In a second step, the architecture optimizer based on optj4 uses an evolutionary algorithm for selecting an initial number of possible topologies (initial population) [Lukasiewycz et al. 2011]. This initial population set is based on the generated model and the available resource containers. Each topology is validated in order to check if all deployment units are at least instantiated once and can communicate with all dependent deployment units via a network connection. The deployment topologies are packaged for simulation including workload and resource profile. We constructed a distributed simulation cluster that can simulate multiple topologies in parallel.

After each simulation run the optimizer evaluates the results. We currently support three optimization goals: (i) approximate the mean resource utilization per resource over of all containers to a certain level (e.g., 70% CPU utilization and 80% memory utilization), (ii) minimize the costs, or (iii) minimize the response time per transaction. A topology can be invalidated if one resource is utilized above a certain threshold to prevent over-utilizing certain containers. The optimizer mutates new topologies based on the evaluation results and delegates the simulation. This process is repeated until the initial population and the number of generations are processed.

The best topology in terms of the optimization goals is a Pareto-front. Therefore, multiple solutions are possible and can be selected by the DevOps engineer. The final step is deploying this topology in the production environment.

5. PERFORMANCE MODEL GENERATOR

This section explains the PMG and extensions we added to support generating a comprehensive performance model. The generated model considers the most important resources: CPU, HDD, network and memory. Furthermore, a transaction flow throughout a DA is detected and used for reasoning the systems control-flow.

We use and extend the PMG of the RETIT Capacity Manager and the corresponding monitoring solution RETIT Java EE. Both solutions are based on the Performance Management Work Tools (PMWT) PMG introduced by Brunnert et al. in 2015 [Brunnert and Krcmar 2015]. As depicted in Figure 3(a), the generation process consists of three phases [Willnecker and Krcmar 2016]:

(i) monitoring the instrumented DA,
(ii) aggregating the monitoring data per operation, and
(iii) generating the performance model based on the aggregated monitoring data.

The result of the three phases is a resource profile, a workload description representing the usage of the DA during the monitoring phase, and a resource environment
describing the current deployment of the DA. All three model parts are stored as a PCM instance [Brunnert and Krcmar 2015; Becker et al. 2009].

5.1. Monitoring

The monitoring step collects operation invocations of the instrumented DA. We distinguish between resource demand measurement and resource demand estimation [Spinner et al. 2015; Willnecker and Krcmar 2016]. Resource demand measurement uses fine-grained monitoring data per operation invocation to measure the exact demand an operation places on a resource. These measurements can be collected with standard APM software like Dynatrace Application Monitoring (AM) [Willnecker and Krcmar 2016]. Resource demand estimation uses coarse-grained monitoring data like total resource utilization and response time series per operation and distributes the utilization throughout the operations [Spinner et al. 2015]. Such coarse-grained resource utilization data can be collected using standard system monitors like System Activity Reporter (SAR), or monitoring and control interfaces of virtual machines like Java Management Extensions (JMX). Load drivers like jMeter or access logs of web servers provide response time series of operations invoked on system-entry level. For more detailed (e.g., component-level) response time series, custom filter or logger are necessary.

The PMG supports data from different data sources as depicted in Figure 3(a):

(i) Application Performance Monitoring for fine-grained application data. We use the RETIT Java EE Monitoring solution in this work. Previous work demonstrated the applicability of industry standard solutions like Dynatrace AM [Willnecker and Krcmar 2016].

(ii) System Performance Monitoring for coarse-grained application data. Standard system tools or custom host agents are possible. We use Apache Webserver access logs, RETIT Host Monitoring and JMX in this work.

The collected data is stored ongoing in a monitoring database based on the Apache Cassandra project. The large amount of data requires a scalable, yet simple database structure. Each row in the database corresponds to an operation invocation or a measurement record from system monitoring. The next phase uses this monitoring database as a single source of input.

---

3http://www.dynatrace.com/
4http://jmeter.apache.org/
5http://httpd.apache.org/
6http://cassandra.apache.org/
Multi-Objective Optimization of Deployment Topologies for Distributed Applications

5.2. Aggregation

The aggregation phase concentrates all the single operation invocations as a preparation for the model generation. The mean demand per resource (e.g., mean CPU demand) is calculated for every operation of every component. Furthermore, the calculation of network demands at the deployment unit boundaries and transaction flows based on unique transaction IDs is conducted in this step.

Figure 3(a) shows the currently supported data sources in the monitoring step; other APM solutions or coarse-grained monitoring providers can be added to the PMG. The demand calculation is either done by the aggregator, or delegated to Library for Resource Demand Estimation (LibReDE) in a post-processing after initial aggregation [Spinner et al. 2015]. After this phase, all operation invocations, their resource demands, and the transaction flow are prepared for the model generation phase.

5.3. Model Generation

This final phase transforms the operation invocations, resource demands, and the transaction flow into a PCM instance. This transformation allows changing and/or simulating, hence predicting the system behavior using the Palladio-Bench [Becker et al. 2009]. PCM supports CPU, HDD and network demands. The built-in features of PCM are used for the above mentioned three resources [Brunnert and Krcmar 2015].

Each operation invocation results in at least one action with a CPU demand. In contrast, for HDD demands we distinguish between write and read demands as the write speed of a HDD is very different from the read speed [Brunnert and Krcmar 2015]. Furthermore, we use the mean request and response size to simulate the time this request travels through the network based on available bandwidth and latency [Brunnert and Krcmar 2015].

Memory demands and simulation are more complex due to automatic memory management [Libič et al. 2015]. We can calculate the mean memory demand of an operation by measuring the memory demand of each operation invocation and forming the average demand similar to CPU demands. A dedicated resource is necessary to place the demands, which supports automatic memory management scenarios as in virtualized runtimes like the Java Virtual Machine (JVM).

We extended the PCM meta-model in order to add a memory resource representation as depicted in Figure 4. This resource works for dynamic memory management scenarios and supports different types of automatic memory management methods, like garbage collections (GCs). GCs delay the release of memory, leading to a larger memory utilization during runtime. Thus, memory is more likely to become a bottle-
neck. We extended PCM to simulate this effect [Becker et al. 2009; Willnecker and Krcmar 2016]. We added two classes to the meta-model to support this behavior:

(i) MemoryResourceSpecification to specify the attributes of a memory resource.
(ii) GarbageCollectionBehaviour to define the behavior of automatic memory management. No behavior is specified in dynamic memory management scenarios.

A typical GC collects and stores released objects in different memory spaces [Libič et al. 2015]. The spaces are cleaned in different intervals. For instance, the JVM executes two types of GCs (minor and major) to clean different spaces or promote objects to another space [Libič et al. 2015]. A memory simulation containing garbage collection requires monitoring GC events and generating instances of the memory resource and the GC behavior in PCM.

For Java EE, we monitor the GC events of the running application using the GarbageCollectorMXBean via JMX and measure the following metrics:

(i) The type of garbage collection that is executed. For Java EE this is either a minor GC or a major GC. Other GC implementations or technologies can have different GC types.
(ii) Size of total memory available in the JVM.
(iii) Size of allocated memory before and after the GC execution. This is a simplification of the actual mechanism as we do not simulate object movements in the fine-grained GC spaces. This probabilistic approach enables automatic memory management simulation with low overhead compared to complex object movement simulations [Libič et al. 2015; Willnecker and Krcmar 2016].
(iv) CPU time necessary to execute the GC.

The measurement data is aggregated and processed for the performance model generation. The resource environment generation creates a MemoryResourceSpecification instance per resource container. This resource contains the initial and the maximum available memory. Furthermore, an optional grow and shrink threshold is part of the resource for simulating changes to the committed memory. For dynamic memory management no further generation is conducted. For automatic memory management we extract the GC types and create a GarbageCollectionBehavior instance for each GC type. For each behavior instance we calculate the mean CPU demand per byte released, the mean free ratio and the threshold leading to a GC execution. Threshold and free ratio are calculated in percent and are independent from the current memory size. This implementation automatically adapts GarbageCollectionBehavior to other resource containers representing larger or smaller servers with less or more available memory for the JVM.

In order to access the newly introduced memory resource we extended ResourceCalls in PCM. We added two more signatures to execute allocation and free calls on this resource. Each operation in the performance model calls the alloc signature of the corresponding memory resource. No free call is necessary for automatic memory management as this is handled by the GarbageCollectionBehavior. In dynamic memory scenarios, the free operation is called after each operation. The available memory is immediately increased by the amount specified in the free call. We extended the PMG to generate such ResourceCalls automatically for every operation that allocates memory.

For automatic memory management a thread per memory resource is started with the simulation. This thread watches if the committed memory of the corresponding resource container exceeds the configured threshold. If the committed memory exceeds the GC execution threshold, a GC run is simulated. The memory of this resource is

7http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/GarbageCollectorMXBean.html
freed depending on the free ratio of the executed \textit{GarbageCollectionBehavior}. A CPU demand depending on the CPU demand per byte of the \textit{GarbageCollectionBehavior} and the number of freed bytes is placed on the CPU resource of the same \texttt{ResourceContainer}.

5.4. Cost model

The costs of on-premise installations are usually flat and thus easy to calculate. The number of servers, their energy consumption, and the maintenance and administration costs are the core influencing factors. In managed infrastructures like IaaS cloud environments, the cost structure gets more complicated. The accounting items are no longer hardware and the staff maintaining it, but uptime of (virtual) instances, different types of network traffic, or average utilization of certain resources. In previous research, we reduced the number of server instances by optimizing the deployment topologies of the software running on it [Willnecker and Krcmar 2016]. We used a simple optimization goal: approximate the resource utilization as close to 70\% as possible. 70\% was selected as higher utilization usually leads to longer and unpredictable response times due to missing/reduced peak tolerance head room. Using 70\% reduced the costs for simple cost structures in on-premise scenarios. Complex cost structures required a more sophisticated method. Therefore, we constructed a flexible cost model for on-premise, cloud and hybrid environments.

Figure 5 depicts our model. We connected the model to the PCM meta-model so that our performance model already contains the cost structure of the target model. Our deployment topology optimizer can now use a holistic cost and performance model and select cost effective deployment topologies for cloud environments.
We connect resource containers (e.g., VMs), resources (e.g., CPUs), and linking resources to a BillingInformation. Thus, each resource or container can be accounted for. The model allows attaching costs based on time using TimingInformation objects. This allows defining the costs over time for resource container for example. The amount of data processed or transferred can be accounted for with the DataInformation class. This is especially useful to model the costs of network traffic. We further distinguish between inbound and outbound traffic. Typical cloud providers offer cheaper costs per Gigabyte (GB) when the data is transmitted within the data centers of the cloud provider (inbound) instead of through the open Internet (outbound).

Several cloud providers have different pricing tiers. The first GBs of network traffic are free, the next couple of GBs are expensive, until the costs decrease for heavy users. To model these tiers, we added a RangeInformation allowing definition of the prices for different tiers. This comprehensive approach allows modeling and calculating the costs of deployment topologies in on-premise and IaaS cloud environments. Therefore, our optimizer can consider resource utilization, response times, and costs when searching for a Pareto-optimal topology of a DA.

6. ARCHITECTURE OPTIMIZER

The core of our deployment topology optimization approach is the architecture optimizer. We use the evolutionary computation algorithms of the opt4j framework (version 3.1.4) [Lukasiewycz et al. 2011]. The technological foundation is similar to design space exploration tools like PerOpteryx, but we use simulations instead of solvers for the evaluation of design alternatives [Koziolek et al. 2011]. We chose opt4j for its collection of optimization algorithms amongst which we use the Evolutionary Algorithm module [Lukasiewycz et al. 2011]. Previous work of Koziolek at el. (2014) and our own experiments produced the best results using this module [Koziolek et al. 2011; Lukasiewycz et al. 2011]. Opt4j in combination with PCM-conform performance models allows us to simulate the performance and cost effects on the topology simulating hundreds of users accessing the system. The simulation approach provides high prediction accuracy although such simulations are computational intensive compared to solver-based approaches. We constructed a simulation cluster to process several simulations in parallel on a dedicated system. This simulation cluster reduces time it takes to find an optimized solution.

The architecture optimizer requires three artifacts to conduct an optimization run as depicted in Figure 3(b):

(i) The resource profile generated by the PMG containing the software architecture and resource demands for all four major resources of the DA.

(ii) The expected workload of the system in the target environment. The topology is optimized according to this workload. In general, the workload in the target environment is expected to be higher compared to the workload executed for the model generation.

(iii) Constraints to the optimization like the minimum/maximum utilization of a resource, the minimum/maximum number of systems, and logical constraints that can, for instance, prohibit deploying the database on a resource container that already contains an application server (AS).

The Optimization Controller acts as workflow controller and triggers all sub components of the system. In a first step the Deployment Topology Allocator creates a random set of valid design alternatives. The allocator considers available resource containers and the network connections between these containers stored in the Resource Container Database. The database consists of a list of all containers available for the optimization run. It can also consist of instances in a local data center or of instances offered by a cloud provider. The capability of the resources of these containers must
be calibrated compared to the resources used during the generation. Brunnert et al. (2015) showed that benchmarks provide accurate results for transforming resource capabilities from one machine to another [Brunnert and Krcmar 2015]. We calibrate network, HDD, and CPU capabilities. This allows us to make performance predictions for an application running in another environment. Therefore, we run HDD, CPU and network benchmarks in both environments and calculate the HDD read and write speed, the relative CPU capability, and the network bandwidth and latency. This calibration has to be done for each host or VM configuration and the network they are connected to.

The initial population of the evolutionary algorithm is calculated by the Deployment Topology Allocator from the list of available containers, deployment units, and constraints. Our population can be represented as a $du \times rc$ matrix $G$. Each column stands for one available resource container $rc$ and each row for a deployment unit $du$ of the DA. The cell values are either 0 or 1. A 1 in the cell $G_{i,j}$ indicates that in this topology the deployment unit $du_i$ is deployed on resource container $rc_j$. The initial topologies are created randomly but invalid selections are discarded immediately. A valid matrix has at least one 1 in each row so that each deployment unit is at least deployed once and fulfills the constraints (e.g., no database (DB) deployment unit on the same container as an AS deployment unit).

The second step is the model-to-model transformation of the matrix into a PCM instance. The matrix representation is transformed into resource environment and allocation model instance. The PMG only creates one instance per deployment unit. Therefore, additional deployment unit instances are created in the repository and system model if necessary during the transformation. The resource environment and allocation model are packaged with the resource profile and workload. The result of this process is a complete PCM instance ready for simulation.

Our evolutionary algorithm evaluates the quality of a topology in multiple components [Łukasiewycz et al. 2011]. First of all, the Optimization Controller checks against the Optimization Database if an equal topology was already simulated. A topology is considered equal if the same deployment units are distributed throughout equal resource containers. Two resource containers are considered equal if their resources have the same capabilities (e.g., same number and speed of the CPU). If an equal topology is detected, the simulation results from a previous run are returned instead of a full simulation run.

The controller dispatches a new simulation job to the simulation cluster if no equal topology has been detected. The cluster consists of a load-balancer and several worker nodes executing the Palladio-Bench in a headless Eclipse instance [Becker et al. 2009]. The load-balancer assigns the simulation job to one worker. If not enough resources for the execution are available, the simulation job is queued. The job is started when resources are free again after, for instance, another simulation job on this worker has been finished. After a job run, the worker stores the results in a shared folder. Each worker node is equally able to simulate a PCM meta-model instance and provide results for already conducted simulations.

The Optimization Controller sends an archive containing all model elements as depicted in Figure 1 to the cluster to start a simulation job. The load-balancer uses a session sticky round-robin approach to balance the load across all simulation workers. This means that new requests will be placed per round-robin on one of the workers. Follow-up requests, like requesting the status or exporting the results, are executed on the same worker node on which the job was started. Furthermore, as the results of the simulation are stored in a shared folder, results can be retrieved from every worker even if the worker that executed the simulation is already shutdown.
The bottleneck of a simulation run is usually the memory resource. Hence, each worker node is memory-aware and only starts a simulation job when enough memory is available. The parameters of a simulation job are part of the initial request and contain the maximum amount of required memory for the simulation. A new job is queued if the maximum required memory of the simulation job exceeds the available memory in the JVM of the worker. To obtain the status of a job the Simulation Results Analyzer queries the cluster using the simulations jobID. The job can either run, be queued, be finished, or has failed. After a job has been finished, the results of the simulation are available as an archive containing all simulation metrics and results.

The Performance Evaluation Tool (PET) is used to analyze the raw results of the simulation and calculates e.g., total resource utilization per resource and container, response times per operation, and total costs [Kross et al. 2016]. The aggregated results together with the topology are stored in the Optimization Database. The controller spawns new topologies based on the evaluation results and the configuration of the evolutionary algorithm [Lukasiewycz et al. 2011]. We currently support three optimization goals:

(i) Approximate the resource utilization of all resources in all used resource containers to a certain level.
(ii) Minimize the total costs of the system.
(iii) Minimize the total response time of the system.

The first optimization goal creates resource efficient topologies. It is important to limit the maximum CPU utilization to 70% or 80%, otherwise the resulting topologies are not representative as the response times become unpredictable due to the high system load. This optimization sorts, for example, memory intensive deployment units to CPU intensive deployment units.

The second optimization goal creates cheaper, more efficient topologies. The number of resource containers here is usually relatively low. Deployment units that interact frequently are deployed on the same server if network traffic is accounted for. The cost optimization goal has a loose relation with the first optimization goal.

For the last optimization goal, we calculate the mean response time over all simulated operations. A deployment topology with a smaller median response time is considered superior to a topology with a larger median response time. This optimization goal tends to require more resources and generates more costs when executed in production.

The applied framework for evolutionary computation supports multi-objective optimization. Hence, our approach supports combinations of the above stated optimization goals. This usually creates multiple optimal solutions along a Pareto-front [Koziolek et al. 2011]. The user must pick one of the solutions or let the optimizer select one solution randomly. Selecting one resulting topology implies accepting trade-offs. Fast topologies are usually more expensive than slower topologies. Depending on the application and its performance requirements, which are not part of the model, one solution might be superior to another. Thus, a manual selection is recommended.

Although the proposed approach attempts to handle deployment topology optimizations in a comprehensive way, it has two limitations. (i) Distributed Database Management Systems (DBMSs) have not been considered. We can today only size VMs for the DBs according to the workload instead of sharding or replicating the DBs. A more advanced DB performance model and dedicated monitoring solutions are required to resolve this limitation. (ii) The results of the architecture optimizer are never certain to be the best solution. Such algorithms can run into local optima and never find the best possible solution or might take extensive time to find the best solution. However, such a structured approach optimizes effectively based on all DA components (workload, resource profile, and available resource containers) [Koziolek et al. 2011].
Table I. Software and hardware configuration for model generation

<table>
<thead>
<tr>
<th>Server</th>
<th>Load-balancer</th>
<th>Driver</th>
<th>Insurance/Vehicle/Provider Server</th>
<th>Insurance/Vehicle/Provider Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Server</td>
<td>Apache 2.2.31</td>
<td>Faban 1.3.0</td>
<td>Wildfly 8.1.0 Final</td>
<td>PostgreSQL 9.4.4</td>
</tr>
<tr>
<td>Database</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Virtual Machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>CentOS 6.7</td>
<td>openSUSE Leap 42.1 (x86_64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Cores</td>
<td>4 vCores (2.1 GHz)</td>
<td>8 vCores (2.1 GHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB</td>
<td>16 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host System</td>
<td>IBM System X3755M3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>1 gigabit-per-second (Gbit/s)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. EVALUATION

7.1. Evaluation System

The SPECjEnterpriseNEXT industry benchmark is the successor of the SPECjEnterprise2010 benchmark. Both are Java EE applications typically used to rate the performance of different Java EE ASs. We use a pre-release version of the SPECjEnterpriseNEXT as example DA for our evaluation. This benchmark mimics an insurance policy management system for car insurances. It consists of three different service components (Insurance, Vehicle, and Insurance Service) and three databases (Insurance, Vehicle, and Provider Database). Furthermore, we added a load-balancer handling requests to the AS instances to enable replicas of the ASs.

The benchmark contains a load driver emulating insurance customers. The so-called Insurance Customer Driver is based on Faban and executes different business transactions on the Insurance Domain, which triggers JAX-RS Representational State Transfer (REST) calls to the other two services and Java Persistence API (JPA) calls to the DBs.

7.2. Evaluation Approach

We conducted two evaluations to demonstrate the capabilities of our approach. One for on-premise installations and one for cloud environments. The second evaluation was conducted in the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) environment to demonstrate the applicability of our approach in industry cloud environments. AWS EC2 environment is the largest public IaaS provider. Applying our research on this providers promises the best leverage to impact industry usage of our approach.

The evaluation process was similar for both environments. In a first step, we deployed a test system in our on-premise environment and executed load on this system to collect monitoring data. The complete configuration is described in Table I. We conducted a run with 100 virtual users to collect APM data. We applied the RETIT Java EE solution as APM software for the ASs. This solution contains a Java Database Connectivity (JDBC) wrapper to collect response times from external DBs. For the load-balancer and database VMs we used the RETIT host monitoring, which collects CPU utilization time series. The load-balancer response times were extracted from Apache HTTP Server access logs. With the response times and utilization we estimated the resource demands for the DB servers and the load-balancer using LibReDE [Willnecker and Krcmar 2016].

Two cost model instances were created. One for each environment considering their different characteristics. The cost-model for the on-premise environment uses only flat costs for running VMs, while the cloud cost-model considers uptime of different VM types, networking costs, and cost-free sections (e.g., first 10 GB traffic free, further

---

8version from 19.02.2016
9http://faban.org/
10http://jax-rs-spec.java.net/
12http://httpd.apache.org/docs/2.2/en/logs.html

ACM Transactions on Internet Technology, Vol. 18, No. 2, Article PRE-PRINT, Publication date: May 2018.
traffic 0.16 ct/GB). We attached the cost models to the generated model. The complete model, an increased workload, and a list of available resource containers were used as input for our deployment topology optimizer.

Afterwards, the optimizer calculated topologies for both environments. The optimizer considered three optimization goals: (i) approximate the utilization of the resources at around 70%, (ii) minimize total costs of operations, and (iii) minimize the median response time over all business transactions. The result of the optimizer was a list of possible deployment topologies. Previous research showed that the optimization algorithm produces accurate results for single optimization goals [Willnecker and Krcmar 2016]. Therefore, we picked one of the best topologies close to the Pareto-front for each environment and deployed it.

Finally, we executed the increased workload on the newly deployed systems. The systems were monitored using system monitoring, which minimizes the influence of the monitoring on the system. Finally, we compared the system monitoring results of a test run in the real environment with the simulation results.

7.3. On-premise Evaluation

We used the topology described in Table I for the model generation run emulating 100 users. After initial generation, we conducted an optimization run discarding all CPU utilization above 70%, minimizing the response times and costs. Furthermore, we increased the workload from 100 to 500 users. The Deployment Topology Allocator created an initial population of 50 different topologies for the evolutionary algorithm. We selected a cross-over rate of 0.95, an offspring population size $\lambda$ of 25 and a parent population size $\mu$ of 25. The algorithm calculated 50 generations resulting in a total of 1250 tested topologies. We used 16 worker nodes in the simulation cluster resulting in about 5 hours of computation.

The resulting topologies and their evaluation criteria are depicted in Figure 7(a). As the costs did not make a huge difference in this scenario we selected a topology with high utilization (meaning less VMs) and relatively good response times. Especially response times can vary from an average response time of 1500 to 2500 ms as depicted on the y-axis in Figure 7(a).
Table II. Measurement and simulation results for a selected topology on-premise accessed by 500 users

<table>
<thead>
<tr>
<th>Resource</th>
<th>Metric</th>
<th>Balancer</th>
<th>AS 1</th>
<th>AS 2</th>
<th>AS 3</th>
<th>AS 4</th>
<th>AS 5</th>
<th>DB 1</th>
<th>DB 2</th>
<th>DB 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>13.84%</td>
<td>71.23%</td>
<td>69.27%</td>
<td>68.19%</td>
<td>72.89%</td>
<td>12.38%</td>
<td>13.61%</td>
<td>65.37%</td>
<td>64.02%</td>
<td>62.87%</td>
</tr>
<tr>
<td>Memory</td>
<td>1 GB</td>
<td>9.30 GB</td>
<td>9.14 GB</td>
<td>4.25 GB</td>
<td>4.76 GB</td>
<td>5.79 GB</td>
<td>6.59 GB</td>
<td>9.79%</td>
<td>7.50 GB</td>
<td>6.32 GB</td>
</tr>
<tr>
<td>HDD</td>
<td>0.24%</td>
<td>1.29%</td>
<td>0.41%</td>
<td>0.69%</td>
<td>1.10%</td>
<td>2.67%</td>
<td>9.22%</td>
<td>10.92%</td>
<td>4.39%</td>
<td></td>
</tr>
</tbody>
</table>

The selected topology required 5 ASs, 3 DB servers and 1 load-balancer. We deployed the system as specified and conducted a run with only system monitoring as instrumentation. Afterwards, we compared the response times and resource utilization from the system monitoring with the simulation results for this topology. The median response time error was about 15% for most of the transactions as depicted in Figure 6. Only the Register and View User business transactions had a larger error of above 20%, still below the 30% acceptable error for capacity planning propagated by Menascé and Almeida (2008) [Menascé 2008].

Table II shows the comparison of simulated and measured resource utilization. The HDD and memory simulation has only been conducted for the ASs due to the fact that LibReDE calculates demands only for the CPU resource and no dedicated monitoring solution was available for these technologies. However, we added our measurements for all resources on all servers, even though memory measurements on the DB servers are only available on process-level. The accuracy is high for CPU demands, especially when LibReDE is used. This confirms previous research comparing LibReDE and resource demand monitoring solutions [Willnecker and Krcmar 2016]. We tend to under-predict CPU demands but over-predict memory demands for the ASs. The under-prediction is a result of overhead tasks of the servers that are not part of the model (e.g., database pool management, load-balancer health checks, etc.) [Willnecker and Krcmar 2016]. The over-prediction of memory demands is a result of the delay between a threshold detection and the GC, the memory grow, or the memory shrink execution in the simulation.

7.4. Cloud Environment Evaluation

We could reuse large parts of the model definition from the on-premise setting. We had to re-calibrate CPU frequencies in the resource environment model as well as network bandwidth and latency. Therefore, we conducted CPU calibrations using the SPEC CPU 2006 benchmark on our local VMs as well as on the different EC2 instances. This allowed us to calculate the relative CPU processing rate between the VMs used for the model generation and the EC2 instances that are simulated. Furthermore, we conducted bandwidth and latency calibrations using lmbench. This sort of calibration allows us to transform the model from one hardware environment to another and provides accurate results even for virtualized environments [Brunnert and Krcmar 2015]. The load driver was still installed in our on-premise environment as a usual customer would access the system from outside of the AWS EC2 environment.

Using the AWS EC2 environment increased the number of possible resource containers (complete M4 instance types except for m4.10xlarge) and regions (e.g., Europe - Ireland, US-East). Therefore, we increase the number of generations calculated by our
optimizer to 100 (from 50 in the previous scenario). This resulted in 2500 tested topologies and took about 10 hours to compute using our 16 simulation worker nodes.

The results of the optimization run are depicted Figure 7(b). As shown, similar response times can be achieved with costs ranging from less than $1500 per month and
Table III. Measurement and simulation results for a selected cloud topology accessed by 500 users

<table>
<thead>
<tr>
<th>Resource</th>
<th>Metric</th>
<th>Balancer</th>
<th>Application Server (AS)</th>
<th>Database (DB)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>AS 1</td>
<td>AS 2</td>
<td>AS 3</td>
<td>DB 1</td>
<td>DB 2</td>
<td>DB 3</td>
</tr>
<tr>
<td></td>
<td>Deployment</td>
<td></td>
<td>Vehicle</td>
<td>Vehicle</td>
<td>Provider</td>
<td>Vehicle</td>
<td>Provider</td>
<td>Vehicle</td>
</tr>
<tr>
<td>CPU</td>
<td>Measured util.</td>
<td>8.78%</td>
<td>38.72%</td>
<td>65.47%</td>
<td>58.99%</td>
<td>56.38%</td>
<td>12.17%</td>
<td>25.38%</td>
</tr>
<tr>
<td></td>
<td>Simulated util.</td>
<td>8.09%</td>
<td>50.91%</td>
<td>39.01%</td>
<td>47.60%</td>
<td>50.07%</td>
<td>12.05%</td>
<td>27.41%</td>
</tr>
<tr>
<td></td>
<td>Relative err.</td>
<td>7.86%</td>
<td>13.32%</td>
<td>14.21%</td>
<td>11.84%</td>
<td>9.59%</td>
<td>0.99%</td>
<td>5.09%</td>
</tr>
<tr>
<td>Memory</td>
<td>Measured dem.</td>
<td>1.02 GB</td>
<td>7.51 GB</td>
<td>5.65 GB</td>
<td>8.09 GB</td>
<td>7.95 GB</td>
<td>8.0 GB</td>
<td>8.0 GB</td>
</tr>
<tr>
<td></td>
<td>Simulated dem.</td>
<td>-</td>
<td>8.13 GB</td>
<td>8.94 GB</td>
<td>8.48 GB</td>
<td>8.46 GB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Relative err.</td>
<td>-</td>
<td>4.10%</td>
<td>4.44%</td>
<td>4.82%</td>
<td>6.42%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HDD</td>
<td>Measured dem.</td>
<td>0.12%</td>
<td>1.37%</td>
<td>1.07%</td>
<td>1.27%</td>
<td>2.31%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Simulated dem.</td>
<td>-</td>
<td>23.36%</td>
<td>18.94%</td>
<td>16.96%</td>
<td>20.07%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Relative err.</td>
<td>-</td>
<td>23.36%</td>
<td>18.94%</td>
<td>16.96%</td>
<td>20.07%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Costs</td>
<td>Measured dem.</td>
<td>116.64</td>
<td>190.09</td>
<td>764.23</td>
<td>764.23</td>
<td>380.38</td>
<td>95.23</td>
<td>380.38</td>
</tr>
<tr>
<td></td>
<td>Simulated dem.</td>
<td>113.97</td>
<td>190.09</td>
<td>764.23</td>
<td>764.23</td>
<td>380.38</td>
<td>95.23</td>
<td>380.38</td>
</tr>
<tr>
<td></td>
<td>Relative err.</td>
<td>2.29%</td>
<td>1.64%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.58%</td>
<td>1.45%</td>
<td>1.58%</td>
</tr>
</tbody>
</table>

up to $3300. We selected a topology with very good response times and upper mid-range costs (about $2800 per month). The response times are depicted in Figure 8. The simulation results are even more accurate compared to the on-premise installation. The relative error of the media response time is below 20% for all cases. The Register and Get Quote business transactions have an error of only 5%. As in the on-premise evaluation, we tend to under-predict the actual response times. Our model lacks certain overhead tasks of the application server and OS. Computation time needed by these tasks are not considered in our model. Therefore, CPU utilization and response times are slightly under-predicted.

The resource utilization results are depicted in Table III. The results are comparable to the previous evaluation. To evaluate the cost estimation, we approximated the costs for a month and compared this with the AWS cost calculator\[^16\], as the calculator also provides costs per month. This included costs for the containers and the outbound traffic. Inbound traffic between the EC2 instances was without cost. The cost estimation error was below 2%. Both the cost and the performance model produce accurate results allowing to predict and thus optimize topologies in cloud environments. Using the network benchmarking even allows prediction of the performance of hybrid environments with components deployed in multiple cloud environments or parts deployed in the cloud and other parts on-premise.

To demonstrate the advantages of our approach, we tested two logical topologies. These topologies were derived by scaling up small instances of AWS EC2 instance and large instances for the second topology. We scaled up deployed 3 m4.large instances until the CPU utilization was below 90% on each instance. This lead to 12 application server instances and 3 DB server instances. The costs were 41.90% higher compared to our selected deployment and the median response times increased by about 135ms. The CPU utilization was at about 68% only one of the DB servers was at about 80% already. To select the second topology we increased the instance type until the CPU utilization was below 90%. This topology used 3 m4.4xlarge machines for the application servers and the same m4.large servers for the DBs. The response times were similar to the response times from the optimized topology but the costs increased by 152.11%. This is due to the relatively small CPU utilization of about 45% on the application servers side. These two topologies use simple strategies but such default topologies are easy to derive as more complex strategies lack of tools to support as the here presented deployment topology optimizer.

\[^16\]http://calculator.s3.amazonaws.com/index.html
8. CONCLUSION

This work successfully connected PMGs and architecture optimizers for optimizing deployment topologies of DAs. This approach allows DevOps engineers to detect the current software architecture and demands of a DA and to size and optimize for on-premise, cloud or hybrid environments. This holds true in up-scaling scenarios as presented in Section 7. This leads to reduced response times, increased resource utilization, and/or reduced costs.

In addition, this work presented an extension for the PCM meta-model for memory resources. The extension supports dynamic and automatic memory management and introduces a probabilistic GC simulation. Monitoring and detecting GC events as well as generating performance models with an accurate memory model have been integrated into the RETIT PMG and the PMWT. The accuracy and feasibility of this approach have been demonstrated.

We presented a flexible cost model as an extension for the PCM meta-model. This cost model can be extended with other billable items and supports tiered pricing models. We demonstrated the accuracy for the AWS EC2 environment with a prediction error below 2%.

In contrast to previously introduced architecture optimizers, our approach uses simulations instead of solvers. The high accuracy of the simulations presented in Section 7 justify this decision. However, this accuracy comes with computational costs. To limit the effect on the decision time we introduced parallel execution in a simulation cluster. We demonstrated the scalability of this cluster with up to 16 worker nodes. Furthermore, optimizations of the simulation process as well as using analytical solvers for a coarse-grained topology estimation before simulating the best candidates would allow to speed up the computation. This would allow to search for more topologies or evaluate the same amount of topologies in shorter period.

Compared to our previous research, we improved the calculation speed by switching from the SimuCom to the EventSim simulation engine [Willnecker and Krcmar 2016]. This increased the amount of simulations that can be conducted by a single worker. Pre-calculating several topologies and selecting the one that best suits the current workload could reduce the disadvantage of computational intensity of our approach. This would allow to use our approach also for runtime decisions.

A continuous monitoring of workload and resource demands in combination with the above mentioned optimizations allow using this approach for runtime decisions. Smart application-aware resource provisioning can increase average resource utilization in data centers, reduce the cost of operations and reduce the carbon footprint due to more efficient typologies. Furthermore, geographic characteristics could be considered during runtime. Latency and thus response times of the application can be reduced by considering the current workload per region and assigning VMs geographically close to the majority of the users. Global workforces can benefit from smart and proactive resource provisioning which takes working hours in different regions of the world into consideration.

REFERENCES


Multi-Objective Optimization of Deployment Topologies for Distributed Applications


