Evaluating the Prediction Accuracy of Generated Performance Models in Up- and Downscaling Scenarios

Andreas Brunnert[1], Stefan Neubig[1], Helmut Krcmar[2]
[1]fortiss GmbH
Guerickestr. 25, 80805 München, Germany
{brunnert, neubig}@fortiss.org
[2]Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract: This paper evaluates an improved performance model generation approach for Java Enterprise Edition (EE) applications. Performance models are generated for a Java EE application deployment and are used as input for a simulation engine to predict performance (i.e., response time, throughput, resource utilization) in up- and downscaling scenarios. Performance is predicted for increased and reduced numbers of CPU cores as well as for different workload scenarios. Simulation results are compared with measurements for corresponding scenarios using average values and measures of dispersion to evaluate the prediction accuracy of the models. The results show that these models predict mean response time, CPU utilization and throughput in all scenarios with a relative error of mostly below 20 %.

1 Introduction

Numerous performance modeling approaches have been proposed to evaluate the performance (i.e., response time, throughput, resource utilization) of enterprise applications [BDMIS04, Koz10, BWK14]. These models can be used as input for analytical solvers and simulation engines to predict performance. Performance models are especially useful when scenarios need to be evaluated that cannot be tested on a real system. Scaling a system up or down in terms of the available hardware resources (e.g., number of CPU cores) are examples for such scenarios.

Evaluating the impact of up- or downscaling on performance is a typical activity during the capacity planning and management processes. Capacity planning concerns questions such as "How many hardware resources are required for the expected workload of new enterprise application deployments?" and involves evaluating the behavior of an application when a system is scaled up. Capacity management on the other hand is usually concerned with evaluating whether the existing hardware resources are sufficient for the current or expected load. This involves not only upscaling but also downscaling scenarios in which the amount of hardware resources needs to be reduced to save costs (e.g., license fees that depend on the number of CPU cores used).

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

Nowadays, creating a performance model requires considerable manual effort [BVD+14]. This effort leads to low adoption rates of performance models in practice [Koz10]. To address this challenge for Java Enterprise Edition (EE) applications, we have proposed an automatic performance model generation approach in [BVK13]. This work improves the existing approach by further reducing the effort and time for the model generation.

In order to evaluate whether the automatically generated performance models are fit for use during capacity planning and management, we evaluate the improved model generation approach in up- and downscaling scenarios. In a first step, an automatically generated performance model is used to predict the performance of a system in an upscaling scenario, in which additional CPU cores are added to the system. Afterwards, a downscaling scenario is evaluated in which the number of CPUs is reduced. During the evaluation of the up- and downscaling scenarios not only the number of CPU cores is modified, but also the amount of users interacting with the system simultaneously.

2 Generating Performance Models

This section is based on our previous work on generating performance models for Java EE applications [BVK13]. In this work, we are using the same concepts for the model generation but reduce the time required for generating a performance model to mostly less than a minute. To make this work self-contained, a brief overview of the model generation process is given, changes are described in more detail. The model generation process is divided into a data collection and a model generation step, the explanation follows these two steps.

2.1 Data Collection

The data that needs to be collected to create a representative performance model is dependent on which components should be represented in the model [BVK13]. Following Wu and Woodside [WW04], Java EE applications are represented using the component types they are composed of. The main Java EE application component types are Applets, Application Clients, Enterprise JavaBeans (EJB) and web components (i.e., JavaServer Pages (JSP) and Servlets) [Sha06]. As Applets and Application Clients are external processes that are not running within a Java EE server runtime, the remainder of this paper focuses on EJB and web components. To model a Java EE application based on these component types, the following data needs to be collected [BVK13]:

1. EJB and web component as well as operation names
2. EJB and web component relationships on the level of component operations
3. Resource demands for all EJB and web component operations

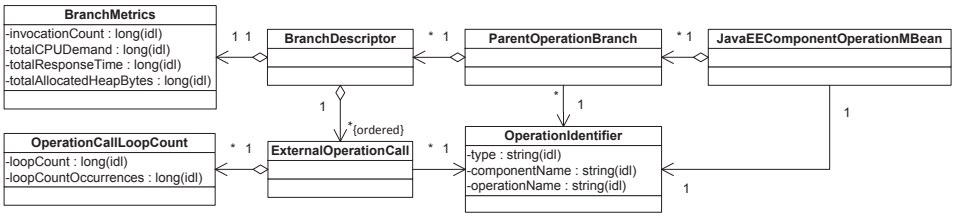


Figure 1: JavaEEComponentOperationMBean data model

In [BVK13] we have collected this information using dynamic analysis, saved it in comma-separated value (CSV) files and used an additional process step to aggregate this information into a database. To speed up the model generation process we are no longer using files as persistence layer and have removed the additional step of aggregating the data stored in the files in a relational database. Instead, the data required for modeling an application is collected and aggregated in Managed Beans (MBeans) [Mic06] of the Java Virtual Machine (JVM). MBeans are managed Java objects in a JVM controlled by an MBean server.

The reason for choosing MBeans as persistence layer is that the Java Management Extension (JMX) specification defines them as the standard mechanism to monitor and manage JVMs [Mic06]. The JMX and related specifications also define ways to manage, access and control such MBeans locally as well as from remote clients. For example, the JMX remote application programming interface (API) allows access to all MBeans of a system remotely using different network protocols. Building upon the JMX standard therefore ensures that the approach is applicable for all products that are compliant with the Java EE [Sha06] and JMX [Mic06] specifications.

One of the key challenges for the transition from CSV files to MBeans is to find a data model with low impact on an instrumented system. The instrumentation for the dynamic analysis collects structural and behavioral information as well as resource demands for each component operation invocation. As storing the data for each invocation separately in an MBean is not possible due to the high memory consumption, the data needs to be aggregated. Additionally, recreation of existing control flows from the data needs to be possible. To accompany these requirements and to implement the MBean data collection with low impact on the monitored system, the data model shown in figure 1 is used.

A *JavaEEComponentOperationMBean* is registered for each externally accessible component operation. Internal component operations are not represented in the data model. Each *JavaEEComponentOperationMBean* instance is identified by an *OperationIdentifier* attribute. The *OperationIdentifier* is a unique identifier of a component operation in a Java EE runtime. It therefore contains the respective *componentName* (i.e., EJB or web component name), the component *type* (i.e., Servlet/JSP/EJB) and its *operationName* (i.e., Servlet/JSP request parameter or EJB method name).

The component relationships are also stored in the data model. These relationships differ depending on component states as well as input and output parameters of their operations (i.e., whether an external operation is called or not). To simplify the data model, com-

ponent states and parameters of component operations are not represented. Instead, the invocation counts for different control flows of a component operation are stored in the model.

A control flow of a component operation is represented by the *BranchDescriptor* class and its ordered list of *ExternalOperationCalls*. *ExternalOperationCalls* are identified using an *OperationIdentifier* attribute. Using this attribute, *ExternalOperationCalls* can be linked to the corresponding *JavaEEComponentOperationMBeans*. This link allows recreating the complete control flows of requests processed by applications in a Java EE runtime.

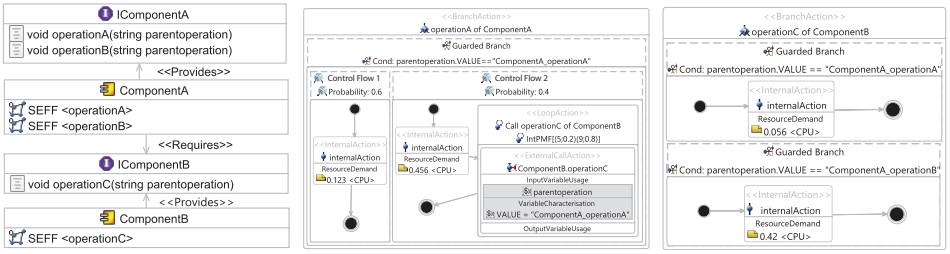
ExternalOperationCalls have an additional *OperationCallLoopCount* attribute, which is used to track the number of times an external operation is called in a row. This attribute helps to limit the amount of data that needs to be stored in the MBeans, as repeating invocations do not need to be stored separately. Instead, each *loopCount* that may occur in an otherwise equal control flow can be tracked using the same data structure. For each *loopCount*, the *OperationCallLoopCount* class stores the number of times a *loopCount* occurred in the *loopCountOccurrences* attribute.

A *BranchDescriptor* also contains a *BranchMetrics* attribute, which tracks the number of times a control flow occurred (*invocationCount*) and how much CPU, heap and response time is consumed by the current component operation in this control flow in total. This information allows calculating the control flow probability and its resource demand during the model generation.

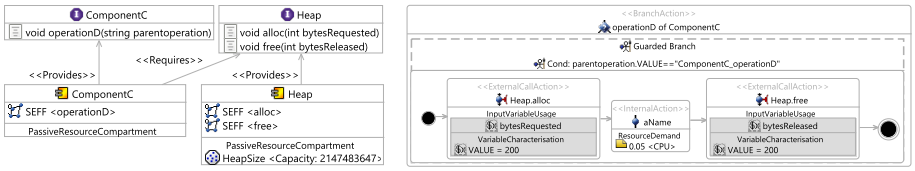
To differentiate requests processed by applications in a Java EE runtime, control flows of an operation are grouped according to the component operation that is invoked first during a request (i.e., by users or external systems). This grouping is specified in the *ParentOperationBranch* class. It maps a list of *BranchDescriptors* to a *JavaEEComponentOperationMBean* and contains a reference to the *OperationIdentifier* of the first operation. This reference improves the data collection and model generation performance.

2.2 Performance Model Generation

The data stored in the MBean data model (see figure 1) is used to generate component-based performance models. The meta model for the generated models is the Palladio Component Model (PCM) [BKR09]. PCM consists of several model layers that are all required to use PCM for performance predictions [BKR09]. This section describes how the repository model layer can be generated automatically as the other model layers can only be created using the information contained in this model. The PCM repository model contains the components of a system, their operation behavior and resource demands as well as their relationships. Repository model components are assembled in a system model to represent an application. User interactions with the system are described in a usage model. The other two model layers in PCM are the resource environment and allocation model. The purpose of a resource environment model is to specify available resource containers (i.e., servers) with their associated hardware resources (CPU or HDD). An allocation model specifies the mapping of components to resource containers. To simplify the use



(a) Repository model example [BVK13] (b) RDSEFF of ComponentA.operationA [BVK13] (c) RDSEFF of ComponentB.operationC



(d) Heap component (e) Calling Heap.alloc and Heap.free from RDSEFFs

Figure 2: PCM repository model elements

of generated repository models, default models for the other PCM model layers are generated automatically once the repository model generation is complete [BVK13]. The PCM usage model is not generated automatically and has to be modeled manually.

Following the data model in figure 1, *JavaEEComponentOperationMBean* instances in a Java EE runtime are used to generate a repository model. The model generation is implemented as a Java-based client that accesses the MBean data using the JMX Remote API. The list of available *JavaEEComponentOperationMBeans* is first of all used to generate component elements in the repository model (e.g., *ComponentA* and *ComponentB* in figure 2(a)). The component list can be derived from the data model by filtering the available *JavaEEComponentOperationMBeans* by the *componentName* attribute of the associated *OperationIdentifier*. Operations provided by Java EE components are also available in the same data structure and are generated in the same step. In a PCM repository model, operations provided by a component are specified in an interface (e.g., *IComponentA* and *IComponentB* in figure 2(a)).

Afterwards, the data in the list of *ExternalOperationCalls* for each *BranchDescriptor* of a *JavaEEComponentOperationMBean* is used to represent component relationships. These relationships are specified in a repository model by a *requires* relationship between the repository component that calls a specific component operation and the interface that provides this operation (e.g., *ComponentA requires IComponentB* in figure 2(a)). The model generation can therefore use information about external operations called in specific operation control flows (using the *OperationIdentifier* of the *ExternalOperationCalls*) to create the relationships of repository model components.

So far, only components, interfaces and their relationships are available in the repository

model. In the next step, the behavior of component operations needs to be specified. The component operation behavior is specified in Resource Demanding Service Effect Specifications (RDSEFF). RDSEFFs are behavior descriptions similar to activity diagrams in the Unified Modeling Language (UML).

As explained in the data collection section 2.1, a component operation can be included in different requests processed by a Java EE runtime (see *ParentOperationBranch* in figure 1). To represent the resulting behavior differences in a performance model, a *parent-operation* parameter is passed between component operations. An example can be found in figure 2(b): *operationA* of *ComponentA* is the first operation called during a request, it thus specifies the *parentoperation* as *ComponentA_operationA* in the external call to *ComponentB.operationC*. This parameter is used in the RDSEFF of *operationC* of *ComponentB* to differentiate the operation behavior depending on the parameter value (see figure 2(c)). The initial *parentoperation* parameter value that is set when a request starts is passed on to all sub-invocations. For example, *ComponentA_operationA* would be passed on if *ComponentB.operationC* would call another external operation within this request. The behavior description of *ComponentA.operationA* in figure 2(b) is also contained in a guarded branch with the condition that the current operation needs to be *ComponentA_operationA*. This is necessary to ensure that all component operations can be used equally in the PCM repository and usage model layers. Thus, operations that only start requests and those that are also (or only) used within requests are indistinguishable.

A component operation can behave differently even though the same *parentoperation* initiated the request processing. These control flow differences are represented in RDSEFFs using probability branches [BKR09]. The probability of each branch (=*BranchDescriptor*) can be calculated based on data in *BranchMetrics* objects. If only one *BranchDescriptor* object exists for a *ParentOperationBranch* object, the probability is one. Otherwise, the *invocationCount* sum of all *BranchMetrics* objects for a *ParentOperationBranch* is used to calculate the probability for a single probability branch in a RDSEFF. An example for such probability branches can be found in figure 2(b). The RDSEFF of *ComponentA.operationA* contains two probability branches (*Control Flow 1* and *Control Flow 2*). One is executed with 60 % probability whereas the second is executed with 40 % probability. The *OperationCallLoopCounts* for different *ExternalOperationCalls* in a specific branch are represented as loop probabilities. For example, in figure 2(b), the external call to *operationC* of *ComponentB* is executed five times in 20 % of the cases and nine times in the other 80 %.

Resource demand data in *BranchMetric* objects is also represented in a probability branch of a RDSEFF. The mean CPU demand in milliseconds (ms) calculated based on the *BranchMetrics* data can be directly assigned to an internal action of a probability branch. In the example in figure 2(b), *ComponentA.operationA* consumes 0.123 ms CPU time in *Control Flow 1*, whereas *Control Flow 2* consumes 0.456 ms.

Representing heap memory demand of a component operation is not directly supported by the PCM meta model. Therefore, the passive resources element of the meta model is reused for this purpose [BKR09]. Even though passive resources are intended to be used as semaphores or to represented limited pool sizes (e.g., for database connections), one can also use them to create a simplistic representation of the memory demand of an application. For this purpose, a heap component is generated in each repository model as shown

in figure 2(d). This heap component contains a specified amount of passive resources that represents the maximum heap size available in a JVM. The maximum configurable value for the available passive resources of the heap component is 2^{31}-1. Thus, if one interprets one passive resource as one byte (B), the maximum configurable heap is two gigabytes (GB). As this is a very low value for Java EE applications nowadays, the model generation can be configured to interpret one passive resource as 10 bytes, so that the maximum representable heap is 20 GB. To do this, all heap memory demands read from the *BranchMetrics* objects are divided by ten and are rounded because passive resources can only be acquired as integer values. As this reduces the accuracy of the model, one passive resource is interpreted as one byte by default.

To allow other component operations in the repository model to consume heap memory, the heap component offers two operations: *alloc(int bytesRequested)* and *free(int bytesReleased)* (see figure 2(d)). This model follows the API for applications written in the programming language C. Using the information about the heap demand gathered in the data collection step (see section 2.1), calls to the *Heap.alloc* operation are generated at the beginning of each execution flow and calls to *Heap.free* at the end. An example is shown in figure 2(e): *operationD* of *ComponentC* calls *alloc* with a value of 200 bytes at the beginning, performs some internal processing and releases the 200 bytes allocated previously. Even though this memory model representation is not realistic for Java applications as the garbage collector (GC) behavior is not represented, the overall utilization of the passive resources helps to get an idea of the heap memory demand of an application.

3 Evaluating the Performance Prediction Accuracy

The feasibility of the model generation approach is evaluated in a case study using a SPECjEnterprise2010[1] industry standard benchmark deployment. SPECjEnterprise2010 is used for this evaluation to make it reproducible as it defines an application, a workload as well as a dataset for a benchmark execution.

3.1 SPECjEnterprise2010 Deployment

The SPECjEnterprise2010 benchmark represents the business case of an automobile manufacturer. It is divided into three application domains. The evaluation in this paper focuses on the Orders domain. This domain is used by automobile dealers to order and sell cars. To avoid the need to model all domains, the communication between the Orders and the other domains is disabled. The setup of the Orders domain consists of a benchmark driver to generate load and a system under test (SUT) on which the Orders domain application com-

[1] SPECjEnterprise is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 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. The official web site for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

ponents are executed. The Orders domain is a Java EE web application that is composed of Servlet, JSP and EJB components. The automobile dealers (hereafter called users) access this application using a web interface over the hypertext transfer protocol (HTTP) and can perform three different business transactions: browse, manage and purchase. These three business transactions are composed of several HTTP requests to the system. The user interactions with the system are implemented as load test scripts in the Faban harness[2]. Faban is a workload creation and execution framework which is used to generate load on the SUT.

The benchmark driver and the SUT are each deployed on a virtual machine (VM) to simplify changing the number of available CPU cores. These two virtual machines are connected by a one gigabyte-per-second network connection and are mapped to an IBM System X3755M3 hardware server which is exclusively used for the SPECjEnterprise2010 benchmarks performed for this evaluation. Both virtual machines run CentOS 6.4 as operating system and are configured to have 20 GB of random-access memory (RAM). The benchmark driver is equipped with eight CPU cores, the number of CPU cores of the SUT is varied during the evaluation. The SPECjEnterprise2010 Orders domain application is deployed on a JBoss Application Server (AS) 7.1.1 in the Java EE 6.0 full profile. The database on the SUT VM is an Apache Derby DB in version 10.9.1.0. The JBoss AS and the Apache Derby DB are both executed in the same JVM, which is a 64 bit Java OpenJDK Server VM in version 1.7.0. An external Java-based client for the model generation is connected to the SUT using the JBoss JMX remote API.

3.2 Evaluating the Data Collection Overhead

The model generation approach introduced in this work relies on data collected using a runtime instrumentation. The instrumentation overhead for collecting the required data is analyzed in this section. As mentioned in the data collection section (see section 2.1), the instrumentation is capable of collecting the CPU and Java heap memory demand for each externally accessible component operation. The instrumentation code is therefore always executed before and after a component operation and can have a great influence on the performance data stored in the performance model.

To evaluate the impact of the data collection, the resource demand of several control flows of the SPECjEnterprise2010 Orders domain application is analyzed using different data collection configurations. For each of the following data collection configurations a SPECjEnterprise2010 benchmark run is executed and afterwards a performance model is generated. The SPECjEnterprise2010 deployment outlined in section 3.1 is used in a configuration with four CPU cores for the SUT. To avoid influences of warm up effects and varying user counts, only steady state data (i.e., data collected during 10 minutes between a five minute ramp up and a 150 second ramp down phase) is collected. The data collection runs are executed with a workload of 600 concurrent users which corresponds to a CPU utilization of the SUT of about 50 %. The resulting performance models contain

[2] http://java.net/projects/faban/

Table 1: Measured instrumentation overhead for the data collection - control flow one

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.sellinventory	1.023 ms	33,650 B	3.001 ms	225,390 B	0.756 ms	3.003 ms
2	CustomerSession.sellInventory	0.785 ms	60,450 B			0.731 ms	
3	CustomerSession.getInventories	0.594 ms	49,540 B			0.548 ms	
4	OrderSession.getOpenOrders	0.954 ms	70,600 B			0.878 ms	
5	dealerinventory.jsp.sellinventory	0.108 ms	16,660 B			0.103 ms	
Total Resource Demand		3.464 ms	230,900 B	3.001 ms	225,390 B	3.015 ms	3.003 ms
Mean Data Collection Overhead		0.116 ms	1378 B			0.003 ms	

the aggregated resource demands collected in the MBeans for these component operations and therefore simplify the analysis. The resource demand for the database is already included in the following measurements, as the embedded derby DB is executed in the same thread as the Servlet, JSP and EJB components.

The mean CPU and heap demands for single component operations involved in three different control flows are shown in tables 1, 2 and 3. Both resource demand types (CPU and heap) are represented as mean values for the data collected during the steady state. The heap demand values in this table are rounded to 10 byte intervals as the model generation is configured to do so to have 20 GB of heap available in the model (see section 2.1).

In a first step, a benchmark run is executed while the CPU and heap demand for all component operations involved in the request processing is collected. The performance model generated based on this configuration is called *Model 1.1* in tables 1, 2 and 3. Afterwards, a benchmark run is executed while only the CPU demand for each component operation is collected. The resulting performance model based on this data collection configuration is called *Model 2.1* in tables 1, 2 and 3. Both benchmark runs are repeated but this time, only resource demand data (CPU or CPU & heap) for the first component operations (those where Order==1 in tables 1, 2 and 3) of each HTTP request is collected. These measurements already include the CPU and heap demands of the sub-invocations (Order $>$1 in tables 1, 2 and 3). The resulting performance models are called *Model 1.2* for the first configuration (CPU and heap collection turned on) and *Model 2.2* for the second configuration (CPU collection turned on).

The total mean CPU and heap demand values in the first model versions (*1.1* and *2.1*) are compared with the corresponding values for the second model versions (*1.2* and *2.2*) to calculate the instrumentation overhead. It can be shown that collecting heap and CPU demands for each component operation is a lot more expensive than only collecting CPU demand. For the HTTP request analyzed in table 1, the mean overhead for the data collection including heap demand is 0.116 ms CPU time and 1378 byte heap memory for each component operation. If only the CPU demand is collected, the mean data collection overhead drops dramatically to 0.003 ms for each component operation.

Other execution flows during the same benchmark runs confirm these values (two additional examples are given in tables 2 and 3). The mean instrumentation overhead for the CPU-only collection is mostly below 0.020 ms whereas the mean instrumentation overhead for the CPU and heap collection ranges mostly between 0.060 and 0.120 ms. As some component operations in the SPECjEnterprise2010 benchmark have an overall CPU

Table 2: Measured instrumentation overhead for the data collection - control flow two

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.view_items	0.406 ms	20,560 B	3.529 ms	615,440 B	0.165 ms	3.566 ms
2	ItemBrowserSession.browseForward	3.315 ms	565,130 B			3.282 ms	
3	ItemBrowserSession.getCurrentMin	0.003 ms	60 B			0.003 ms	
4	ItemBrowserSession.getCurrentMax	0.003 ms	60 B			0.002 ms	
5	ItemBrowserSession.getTotalItems	0.003 ms	60 B			0.002 ms	
6	purchase.jsp.view_items	0.147 ms	40,380 B			0.142 ms	
Total Resource Demand		3.877 ms	626,250 B	3.529 ms	615,440 B	3.598 ms	3.566 ms
Mean Data Collection Overhead		0.070 ms	2162 B			0.006 ms	

Table 3: Measured instrumentation overhead for the data collection - control flow three

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.add_to_cart	0.213 ms	11,300 B	0.393 ms	27,520 B	0.108 ms	0.388 ms
2	OrderSession.getItem	0.276 ms	13,460 B			0.255 ms	
3	shoppingcart.jsp.add_to_cart	0.059 ms	5960 B			0.058 ms	
Total Resource Demand		0.548 ms	30,720 B	0.393 ms	27,520 B	0.421 ms	0.388 ms
Mean Data Collection Overhead		0.077 ms	1600 B			0.017 ms	

demand of below 0.150 ms, collecting the heap demand for this deployment causes too much overhead. The following evaluation therefore focuses on models generated based on the CPU demand collection.

3.3 Comparing Measured and Simulated Results

In the next two sections, the prediction accuracy of generated performance models is evaluated in an upscaling and a downscaling scenario. The steps for both evaluations are similar and are described in the following paragraphs.

Load is generated on the SUT to gather the required data for the model generation in each scenario using the data collection approach outlined in section 2.1. As the database is included within the server JVM, the collected data already contains its CPU demands. Similar to the benchmark runs in the overhead evaluation, only steady state data (i.e., data collected during 10 minutes between a five minute ramp up and a 150 second ramp down phase) is collected. Afterwards, a software prototype that implements the performance model generation approach is used to generate a PCM model based on the collected data.

PCM models can be taken as the input for a simulation engine to predict the application performance for different workloads and resource environments. The standard simulation engine for PCM models is SimuCom which uses model-2-text transformations to translate PCM models into Java code [BKR09]. The code is then compiled and executed to start a simulation. To evaluate the accuracy of the simulation results, they are compared with measurements on the SUT. The following comparisons only use steady state data collected during simulation and benchmark runs of similar length.

The benchmark driver reports the mean response time and throughput for each business

transaction for a benchmark run. However, the predicted response time values cannot be compared with response time values reported by the driver, because they do not contain the network overhead between the driver and the SUT. Therefore, response time of the business transactions browse (B), manage (M) and purchase (P) is measured on the SUT using an additional instrumentation. To identify business transactions using this instrumentation, the benchmark driver is patched to add a unique transaction identifier to each request. This identifier allows combining several HTTP requests into one business transaction. Incoming requests are aggregated on the fly to the business transaction they belong to by summing up their response times. The resulting business transaction response time measurements are stored with a timestamp to calculate the mean throughput on a per-minute basis.

The CPU time consumed by the JVM process of the JBoss AS on the SUT (and thus its CPU utilization) is collected every second to reconstruct its approximate progression and to compare the measured and simulated ranges. The calculation of the mean CPU utilization is based on the first and the last CPU time consumption value in the steady state of a benchmark run in order to avoid biasing effects caused by unequal measurement intervals.

Each benchmark run is performed three times, the results are combined giving each run the same weight. Since all runs have the same duration, the overall mean value of CPU utilization can be calculated by averaging the corresponding values of each run. The throughput values represent the amount of times a business transaction is invoked per minute, thus the collected per-minute values are combined to a mean value. To evaluate response times, samples of equal sizes are drawn from each result. Response time measurement and simulation results are described using mean and median values as well as values of dispersion, namely the quartiles and the interquartile range (IQR). Variance and standard deviation are excluded from our investigation due to the skewness of the underlying distributions [Jai91] of the response times of browse, manage and purchase. In the following sections, means are illustrated tabularly, medians and quartiles are illustrated using boxplot diagrams.

3.4 Evaluating Prediction Accuracy in an Upscaling Scenario

To evaluate the performance prediction accuracy of automatically generated performance models in an upscaling scenario, the number of CPU cores for simulation and benchmark runs is increased step by step. To increase the CPU core count of the SUT for the benchmark runs, the VM is reconfigured accordingly. The number of simulated CPU cores is varied by editing the generated resource environment model.

If workload stays stable, the CPU utilization significantly declines with each increase of cores as does its impact on the overall application performance. As a result, after reaching a sufficient number of CPU cores, the measured response times stay almost constant regardless of any further increases while the simulated response times decrease further reaching their lower bound only at a very high number of CPU cores. Therefore, an increasing inaccuracy in the simulated values is expected since the generated model solely

Table 4: Measured and simulated results in an upscaling scenario

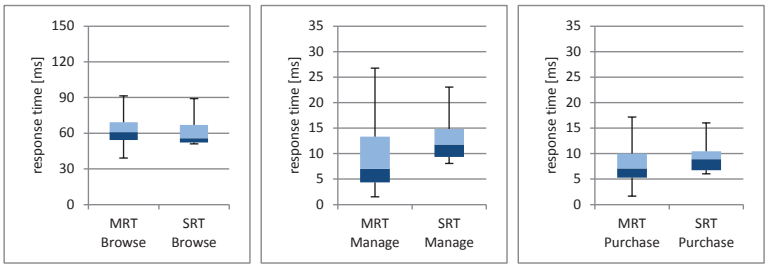
C	U	T	MMRT	SMRT	RTPE	MMT	SMT	TPE	MCPU	SCPU	CPUPE
4	600	B	63.23 ms	65.06 ms	2.91 %	1820.6	1813.1	0.41 %	48.76 %	46.87 %	3.88 %
		M	11.58 ms	13.28 ms	14.71 %	906.8	917.3	1.16 %			
		P	8.27 ms	9.73 ms	17.67 %	904.9	900.3	0.50 %			
6	900	B	69.25 ms	57.56 ms	16.89 %	2708.3	2721.5	0.49 %	51.72 %	46.85 %	9.42 %
		M	12.54 ms	11.95 ms	4.69 %	1354.3	1354.4	0.01 %			
		P	8.95 ms	8.72 ms	2.60 %	1352.4	1368.1	1.16 %			
8	1200	B	88.82 ms	56.25 ms	36.66 %	3617.8	3641.9	0.67 %	57.34 %	46.97 %	18.09 %
		M	14.13 ms	11.64 ms	17.67 %	1806.4	1795.0	0.63 %			
		P	9.31 ms	8.46 ms	9.15 %	1811.6	1819.2	0.42 %			

depends on CPU demands and disregards other factors such as I/O operations on hard disk drives. Thus, to keep the CPU utilized, the workload on the system is varied proportional to the number of CPU cores by increasing the number of concurrent users accessing the SUT. In the following, a performance model generated on the SUT configured with 4 CPU cores is used. The average CPU utilization while gathering the data required for the model generation was 52.46 % which corresponds to a closed workload consisting of 600 users with an average think time of 9.9 s.

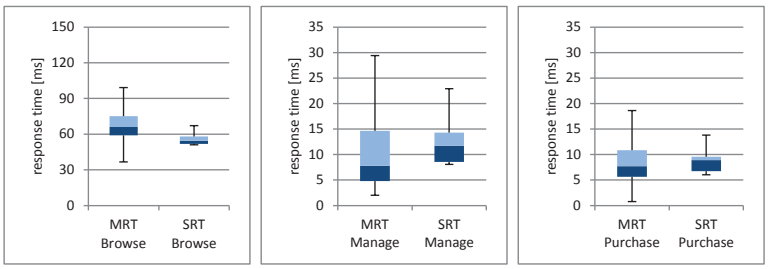
In a first step, the generated model is evaluated by simulating the application performance for an environment which is equal to the one the model has been generated with. Afterwards, the model is evaluated for environments with an increased number of CPU cores. The measured and simulated results are shown in table 4. For each configuration specified by the number of cores (C) and the number of users (U), the table contains the following data per business transaction (T): Measured Mean Response Time (MMRT), Simulated Mean Response Time (SMRT), relative Response Time Prediction Error (RTPE), Measured Mean Throughput (MMT), Simulated Mean Throughput (SMT), relative Throughput Prediction Error (TPE), Measured (MCPU) and Simulated (SCPU) Mean CPU Utilization and the relative CPU Utilization Prediction Error (CPUPE).

The simulation predicts the mean response time of the business transactions with a relative error of less than 20 %, except for the browse transaction in the case of 8 CPU cores and 1200 concurrent users, which shows a relative prediction error of 36.66 %. CPU utilization is predicted with relative errors ranging from 3.88 % to 18.09 %. Due to space limitations, the span consisting of the minimum and maximum of the measured and simulated CPU utilization values is not shown. However, while both ranges mostly overlap, the measured span lies slightly above the simulated one. The same applies to the mean CPU utilization values shown in table 4, as the simulated mean is slightly lower than the measured one. The prediction of the mean throughput is very close to the real values, as the think time of 9.9 s is much higher than the highest response time. Response time prediction errors thus have a low impact on the throughput. Except for the last simulation of browse, the quality of the predictions ranges from very good to still acceptable for the purpose of capacity planning [MAL+04].

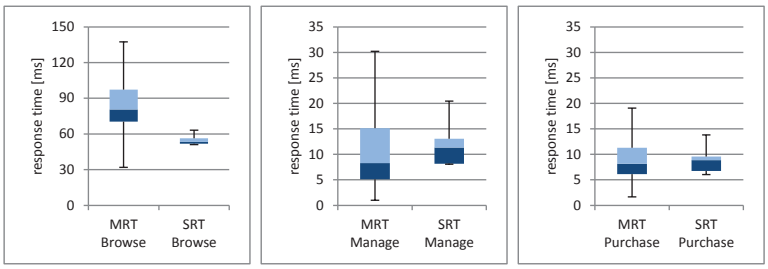
Further statistical measures are illustrated as boxplot diagrams in figure 3. Boxplot diagrams consist of a box whose bounds denote the first quartile Q_1 (lower bound) as well as the third quartile Q_3 (upper bound) of the underlying data sample. The quartiles are con-



(a) 4 CPU cores and 600 users



(b) 6 CPU cores and 900 users



(c) 8 CPU cores and 1200 users

Figure 3: Boxplot diagrams of an upscaling scenario

nected by vertical lines to form the box that indicates the interquartile range (IQR) which is defined as $Q_3 - Q_1$. Furthermore, the median Q_2 is illustrated by a horizontal line within the box, thus separating it into two parts. Vertical lines outside the box (whiskers) indicate the range of possible outliers while their length is limited to 1.5 times the IQR.

The relative prediction error of the median response time ranges from 8.38 % to 33.80 % for the browse and purchase transactions. The median response time of the manage transaction, however, is predicted with a relative error of 36.52 % to 68.29 %. The skewness of a business transaction's underlying distribution can be determined considering the median's position between the quartiles Q_1 and Q_3. The boxplot diagrams in figure 3 show that the skewness is not simulated correctly. To investigate the dispersion of business transactions, we determine the IQR. Its relative prediction error ranges from 21.96 % to 50.94 % for the

Table 5: Measured and simulated results in a downscaling scenario

C	U	T	MMRT	SMRT	RTPE	MMT	SMT	TPE	MCPU	SCPU	CPUPE
8	800	B	71.54 ms	64.03 ms	10.50 %	2413.9	2415.8	0.08 %	37.41 %	35.17 %	5.99 %
		M	12.96 ms	12.64 ms	2.49 %	1203.5	1209.2	0.48 %			
		P	9.36 ms	9.33 ms	0.25 %	1215.9	1228.7	1.05 %			
6	800	B	67.62 ms	66.03 ms	2.35 %	2413.9	2425.4	0.48 %	46.38 %	46.94 %	1.21 %
		M	12.52 ms	13.08 ms	4.45 %	1202.0	1196.6	0.45 %			
		P	9.05 ms	9.64 ms	6.57 %	1208.2	1215.0	0.56 %			
4	800	B	71.15 ms	87.46 ms	22.92 %	2437.0	2420.8	0.66 %	65.60 %	70.27 %	7.12 %
		M	12.98 ms	17.04 ms	31.29 %	1199.7	1193.5	0.51 %			
		P	8.93 ms	12.88 ms	44.33 %	1211.6	1212.1	0.04 %			

manage and purchase transactions and is up to 83.13 % for the browse transaction.

In the measurement results, the effect of increasing the workload dominates, thus the measured CPU utilization slightly increases from 48.76 % to 57.34 %. The response times increase accordingly. In the simulation results, the effect of core increase slightly dominates over the effect of increasing the workload. Therefore, the response times slightly decrease over the course of the experiment, while the simulated CPU utilization remains almost constant.

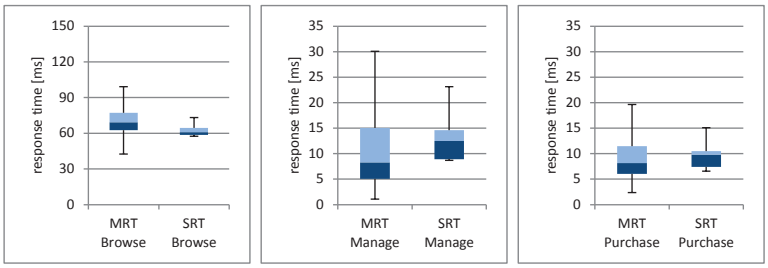
3.5 Evaluating Prediction Accuracy in a Downscaling Scenario

The prediction accuracy of generated performance models in a downscaling scenario is evaluated by reducing the number of CPU cores step by step. Starting with 8 CPU cores, the number of cores is decreased by 2 in each evaluation step. This scenario does not require the number of users to be varied, as the CPU utilization increases. The business case of scaling the number of CPU cores down is to optimize production systems (e.g., to evaluate if several applications can be hosted on one machine or to reduce license fees). In this case, the number of users does not change. Therefore, the workload is kept constant at 800 users with a think time of 9.9 s accessing the SUT in parallel.

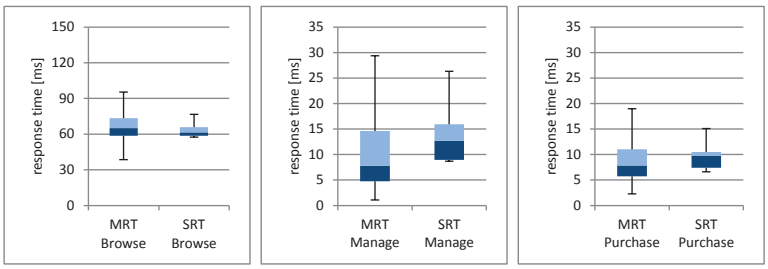
Since the number of cores is reduced during the experiment, a sufficiently low starting value of CPU utilization is required. Therefore, data to generate a performance model for this evaluation is collected with an average CPU utilization of 38.9 %. To compare the simulation results with the measured ones, the previously described evaluation process is applied. The comparison of the mean response time, CPU utilization and throughput values is shown in table 5.

The relative prediction error for the mean response time of all business transactions is at most 44.33 %. CPU utilization is predicted with a maximum relative error of 7.12 %. In contrast to the upscaling scenario, the simulated CPU utilization grows slightly above the measured results as the CPU cores are decreased. The relative prediction error of the mean throughput is about 1 %.

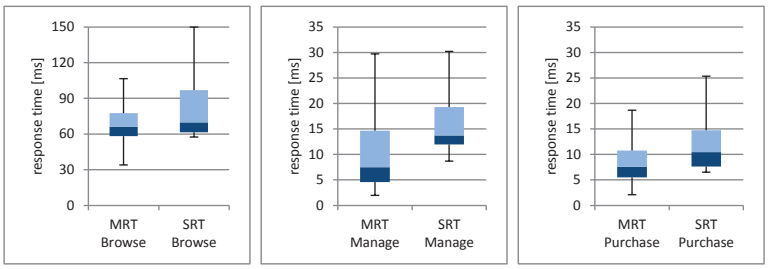
The relative prediction error of the median response time as shown in the boxplots in figure 4 ranges from 5.27 % to 38.50 % for the browse and purchase transactions and from



(a) 8 CPU cores and 800 users



(b) 6 CPU cores and 800 users



(c) 4 CPU cores and 800 users

Figure 4: Boxplot diagrams of a downscaling scenario

50.51 % to 82.96 % for the manage transaction. This is in line with the observations previously made in the upscaling scenario. The relative IQR prediction error ranges from 27.34 % to 43.48 % for the manage and purchase transaction; for the browse transaction it is up to 83.37 %.

Comparing the measured mean and median response times shows that the lowest values are achieved in the 6 CPU core configuration. Even with 4 CPU cores, the browse and purchase response times are lower than in the 8 CPU core configuration. As the CPU utilization of the investigated configurations is relatively low, the lower performance of the SUT with 8 CPU cores can be explained by an increased scheduling overhead. Due to the low CPU utilization, its impact on the overall performance of the SUT is lower than the impact of other factors such as I/O operations. The response time prediction behaves

incorrectly in these cases, as the generated performance model only relies on the CPU demand measured during the data collection step and does not take these effects into account. However, the simulation of CPU utilization is still very close to the measurements. This is useful for determining a lower bound of feasible configurations regarding the amount of cores. Simulating an environment consisting of 3 CPU cores results in a simulated CPU utilization of 91.96 % and indicates that this would lead to instability of the SUT for the given workload. This configuration is thus not investigated in this downscaling scenario.

4 Related Work

Running Java EE applications have already been evaluated using performance models by several authors. Chen et al. [CLGL05] derive mathematical models from measurements to create product-specific performance profiles for the EJB runtime of a Java EE server. These models are intended to be used for performance predictions of EJB components running on different Java EE products. Their approach is thus limited to Java EE applications that solely consist of this component type.

Liu et al. [LKL01] also focus on EJB components and show how layered queuing networks can be used for the capacity planning of EJB-based applications. In their work, they model an EJB-based application manually and describe how an analytical performance model needs to be calibrated before it can be used for the capacity planning. To improve this manual process, Mania and Murphy [MM02] proposed a framework to create analytical performance models for EJB applications automatically. However, the framework was never evaluated to the best of our knowledge.

The difficulties in building and calibrating performance models for EJB applications manually are also described by McGuiness et al. [MML04]. Instead of using analytical solutions to predict the performance of an EJB-based application, they are using simulation models. The authors argue that simulation models are better suited for the performance evaluation of EJB applications due to their flexibility and increased accuracy compared to analytical models.

The applicability of analytical performance models for Java EE applications with realistic complexity is analyzed by Kounev and Buchmann [KB03] using the SPECjAppServer2002 industrial benchmark. Kounev extends this work in [Kou06] by using queuing Petri nets to evaluate the performance of a SPECjAppServer2004 benchmark deployment. The latest version of the SPECjAppServer benchmark (SPECjEnterprise2010) is used by Brosig and Kounev in [BHK11] to show that they are able to semi-automatically extract PCM models for Java EE applications. Their model generation approach is based on data generated by the monitoring framework of Oracle's WebLogic product and thus not transferable to other Java EE server products. It also requires manual effort to distribute the resource demand based on the service demand law once a model is generated.

The previous work is extended by the approach introduced in this work as it is applicable for all Java EE server products and can generate performance models for EJB as well as for web components automatically.

5 Conclusion and Future Work

The approach presented in this work aims to make performance modeling better applicable in practice. The ability to generate performance models at any time simplifies their use in Java EE development projects, as the effort to create such models is very low. The evaluation showed that the generated performance models predict the performance of a system in up- and downscaling scenarios with acceptable accuracy. The approach can thus support related activities during the capacity planning and management processes.

Future work for this approach includes extending the data collection and model generation capabilities. First of all, we need to investigate whether the user session information available in the Java EE runtime can be used to generate usage models automatically. Further extensions are required to support additional technologies specified under the umbrella of the Java EE specification, such as JavaServer Faces (JSF) or web services. For this purpose, the model generation approach also needs to be extended to support distributed systems. A key challenge for such an extension is the integration and correlation of MBean data collected from multiple Java EE servers. Additional improvements are required to reduce the instrumentation overhead as soon as heap demand needs to be collected.

6 Acknowledgements

The authors would like to thank Jörg Henß, Klaus Krogmann and Philipp Merkle from the Karlsruhe Institute of Technology (KIT) and FZI Research Center for Information Technology at KIT for their valuable input and support while implementing the heap representation approach in PCM.

References

[BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[BHK11] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, pages 183–192, Oread, Lawrence, Kansas, USA, 2011.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BVD+14] Andreas Brunnert, Christian Vögele, Alexandru Danciu, Matthias Pfaff, Manuel Mayer, and Helmut Krcmar. Performance Management Work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[BVK13] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In Maria Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2013.

[BWK14] Andreas Brunnert, Kilian Wischer, and Helmut Krcmar. Using Architecture-Level Performance Models As Resource Profiles for Enterprise Applications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 53–62, New York, NY, USA, 2014. ACM.

[CLGL05] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance Prediction of Component-based Applications. *Journal of Systems and Software*, 74(1):35–43, 2005.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley Computer Publishing, John Wiley & Sons, Inc., 1991.

[KB03] Samuel Kounev and Alejandro Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proceedings of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG), Dallas, Texas, USA*, pages 273–283, 2003.

[Kou06] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[LKL01] Te-Kai Liu, Santhosh Kumaran, and Zongwei Luo. Layered Queueing Models for Enterprise JavaBean Applications. In *Proceedings of the IEEE International Conference on Enterprise Distributed Object Computing*, pages 174–178, Washington, DC, USA, 2001. IEEE.

[MAL+04] Daniel A. Menascé, Virgilio A. F. Almeida, F. Lawrence, W. Dowdy, and Larry Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall, Upper Saddle River, New Jersey, 2004.

[Mic06] Sun Microsystems. Java Management Extensions (JMX) Specification, vers. 1.4, 2006.

[MM02] D. Mania and J. Murphy. Framework for Predicting the Performance of Component-Based Systems. In *Proceedings of the 10th IEEE International Conference on Software, Telecommunications and Computer Networks*, Croatia, Italy, 2002.

[MML04] D. McGuinness, L. Murphy, and A. Lee. Issues in Developing a Simulation Model of an EJB System. In *Proceedings of the 30th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems*, Las Vegas, Nevada, USA, 2004.

[Sha06] Bill Shannon. Java Platform, Enterprise Edition (Java EE) Specification, v5, 2006.

[WW04] Xiuping Wu and Murray Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.