Detecting Performance Change in Enterprise Application Versions Using Resource Profiles

Andreas Brunnert
fortiss GmbH
Guerickestr. 25
80805 München, Germany
brunnert@fortiss.org

Helmut Krcmar
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
krcmar@in.tum.de

ABSTRACT
Performance characteristics (i.e., response time, throughput, resource utilization) of enterprise applications change for each version due to feature additions, bug fixes or configuration changes. Therefore, performance needs to be continuously evaluated to detect performance changes (i.e., improvements or regressions). This work proposes a performance change detection process by creating and versioning resource profiles for each application version that is being built. Resource profiles are models that describe the resource demand per transaction for each component of an enterprise application and their control flow. Combined with workload and hardware environment models, resource profiles can be used to predict performance. Performance changes can be identified by comparing the performance metrics resulting from predictions of different resource profile versions (e.g., by observing an increase or decrease of response time). The source of changes in the resulting performance metrics can be identified by comparing the profiles of different application versions. We propose and evaluate an integration of these capabilities into a deployment pipeline of a continuous delivery process.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: measurement techniques, modeling techniques

General Terms
Measurement, Performance

Keywords
Performance Evaluation, Performance Change Detection, Palladio Component Model, Java, Enterprise Applications

1. INTRODUCTION
Performance characteristics of enterprise applications change whenever new features or fixes are introduced during software development [9]. Evaluating the performance impact of such changes nowadays requires a performance test which consumes a lot of time and resources. Due to the associated effort and cost, performance tests are often not executed for each version. To improve the feedback cycle during software development, we propose a performance change detection process for each application version that is being built.

This work proposes the use of resource profiles to realize the performance change detection process. The term resource profile is used to describe the resource demand per transaction of an enterprise application version [3, 11]. It typically includes central processing unit (CPU) usage, disk IO traffic, memory consumption, and network bandwidth for each component of a software system [3, 11]. In [6] we introduced a concept to use sub-models of an architecture-level performance meta-model as resource profiles. These model-based resource profiles provide a better separation of transaction resource demands from the workload and hardware environment compared to the traditional way of specifying resource profiles using vectors [3]. They specify the resource demand of different transactions, but also the components involved in the transaction processing as well as their control flow. It is furthermore possible to predict performance (i.e., response time, resource utilization, and throughput) using these model-based resource profiles by extending them with workload and hardware environment models [13].

Following the definition of Cherkasova et al. [8], performance changes are defined as an increase or decrease of transaction processing time. Using predictions of different resource profile versions for the same workload and hardware environment model(s), performance changes can be identified by comparing the resulting response time prediction results. If a change is detected, resource profiles can be compared with each other to support the identification of the root cause. To realize such a performance change detection process, a resource profile version needs to exist for each enterprise application version that is being built. Due to this reason, we propose to integrate this change detection process into a deployment pipeline of a continuous delivery process as outlined in the next section.

2. DETECTING PERFORMANCE CHANGE WITHIN A DEPLOYMENT PIPELINE
Continuous integration (CI) systems integrate independently developed code provided by different teams and build the overall system composed of all components involved. CI systems help to keep the development teams in sync and to

VALUETOOLS 2014, December 09-11, Bratislava, Slovakia
Copyright © 2015 ICST 978-1-63190-057-0
DOI 10.4108/icst.valuetools.2014.258184

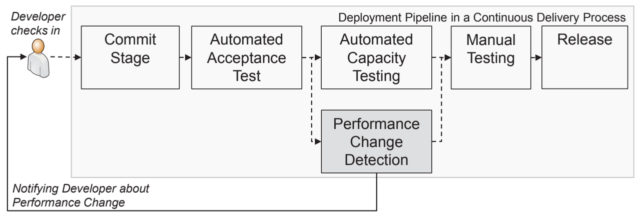

Developer checks in
Deployment Pipeline in a Continuous Delivery Process
Commit Stage
Automated Acceptance Test
Automated Capacity Testing
Manual Testing
Release
Performance Change Detection
Notifying Developer about Performance Change

Figure 1: Detecting performance change within a deployment pipeline (adapted from [9])

create deployable artifacts automatically. CI systems such as Jenkins[1] are very popular nowadays as code changes that break the build or cause problems in automated unit tests can be identified immediately. This immediate feedback of potential problems increases the awareness of everyone involved in the development process regarding the impact of a change on the overall system.

As compiling and building a software system is only the first step to get a system in a state so that it can be used by its end users, a new discipline called continuous delivery (CD) emerged in recent years [9]. CD is defined by Humble and Farley [9] as an extension of CI principles to the "last mile" to operations. A key element in their definition of CD is a so called deployment pipeline. A deployment pipeline describes the steps it takes from a deployable version until a program version exists that can be used as a release candidate.

Figure 1 depicts an extended version of the deployment pipeline defined in [9]. As a first step in this deployment pipeline, a build is triggered by one or multiple developer check-ins. These check-ins are used by a CI system in a so called commit stage to build a new version of an application and to execute a set of predefined tests. These tests focus on low-level application programming interface (API) tests from a developer perspective. The next stage, called automated acceptance test, evaluates if an application that is being built really delivers value to a customer. A set of regression tests is being executed to evaluate the functionality from an end-user perspective. Humble and Farley [9] suggest that afterwards an automated capacity test should occur. If the capacity test results are acceptable, the new version is being tested manually before it is released as candidate for production deployment. It is important to note, that continuous delivery does not imply continuous deployment, and does not automatically deploy a release candidate to production.

This work proposes a performance change detection process using resource profiles as an alternative to the capacity testing step within such a deployment pipeline (see figure 1). Such an integration not only helps to ensure that a resource profile is created for every build but also to identify the cause of a performance change. As the source code changes included in a specific build are known, they can be specifically analyzed whenever a performance change is detected. The steps of the performance change detection process are outlined in the following section.

[1]http://jenkins-ci.org/

2.1 Performance Change Detection

In their description of the capacity tests within the deployment pipeline, Humble and Farley [9] define capacity as the maximum throughput a system can achieve with given response time thresholds for different transaction types. This definition implies that a system needs to be available during the automated capacity testing step which is comparable to the final production environment. Otherwise, results from capacity tests according to their capacity definition would not yield meaningful results. This precondition is often not given in development projects [4], because representative test environments are shared between projects to reduce cost. The immediate feedback that is promised by CD systems can, thus, often not be guaranteed using this approach, as only smaller scale systems are available. The capacity test results derived from such systems are hardly comparable to a production environment.

Using resource profiles, performance can be predicted for hardware environments with more resources (e.g., CPUs) than the ones that are available [6]. Furthermore, resource profiles can be adapted to a hardware environment which is more comparable to the production environment using an approach presented in [6]. These abilities speed up the feedback loop and allow for performance evaluations that would be otherwise impossible. Real capacity tests can then be executed later in the process or as a manual test step, when appropriate systems are available.

Detecting performance change using resource profiles requires the following steps: In a first step, a resource profile for the current application version needs to be created. This resource profile version must be put into a versioning repository to make it available for subsequent builds. Afterwards, performance is predicted using the current version and the results are compared with prediction results from a previous version to see if a performance change occurred. If a change is detected, a notification is sent to the development team(s). The next sections explain how these steps are realized and integrated into a continuous delivery process.

2.2 Creating Resource Profiles

We propose the use of dynamic analysis to collect measurements during the automated acceptance test execution to create resource profiles. This approach has several advantages, apart from the fact that it saves time compared to a separate measurement run (according to Humble and Farley [9], acceptance tests usually take several hours to complete): Transactions that are being tested and executed in the acceptance test stage are expected to be close to the behavior of the users [9]. Using measurements from the acceptance tests also ensures that the workload stays relatively stable as the regression tests are executed for every build. The measurement results for each build are, therefore, comparable.

As mentioned in the introduction, a model-based resource profile of an enterprise application is represented using architecture-level performance models [6]. We use the Palladio Component Model (PCM) as meta-model to represent resource profiles [2]. The PCM meta-model represents the performance-relevant aspects of a software system separately from the workload and the hardware environment. Performance-relevant aspects of a software system are represented in a so called repository model. This model contains components of a software system and their relationships. The control flow of a component operation, its resource demand and parametric dependencies are also specified in this model. Components are assembled in a system model to represent an application. We use the system model to group components by the deployment units they belong to, to simplify their use [6]. The workload on a system is described in a usage model. The remaining two model types in PCM describe the hardware environment: A resource environment model allows to specify available resource containers (i.e., servers) with their associated hardware resources (e.g., CPU or HDD). An allocation model specifies the mapping of system model elements on resource containers. A resource profile for an enterprise application can thus be represented by a repository together with a system model [6].

Several ways to generate these two PCM model types either based on static [1] or dynamic analysis [5] exist. We do not explain a specific model generation approach in this section to focus on the conceptual use of resource profiles for the purpose of detecting performance change[2]. However, for the purpose of identifying performance change, it is only feasible to use approaches that use measurement data from dynamic analysis. Otherwise, the required resource demand values for representing a resource profile would be missing [17].

2.3 Versioning Resource Profiles

Each reusable artifact that is created within the deployment pipeline is stored in a so called artifact repository [9]. This is necessary to make each artifact available in different steps of the deployment pipeline. To detect performance change, we do not only need the resource profile of the current version but also the one of previous builds. Therefore, each resource profile version that is created in the deployment pipeline is stored in an artifact repository that allows to manage different resource profile versions.

As PCM is based on the Eclipse Modeling Framework (EMF)[3], all performance models conform not only to the PCM meta-model but also to the Ecore meta-model defined by EMF. We are leveraging this capability by using the EMFStore [12], which already implements the required versioning features for models based on the Ecore meta-model.

The advantage of using EMFStore compared to other versioning systems is that it is especially designed to support the semantic versioning of models [12]. Instead of working with textual representations of the models in existing systems, EMFStore uses the Ecore model elements and their relationships to manage models stored in the repository. For example, instead of representing a structural change between two model versions as multiple lines in their textual representation, EMFStore directly stores the change in the Ecore model itself [12]. The only two PCM model layers that are currently stored and versioned in the EMFStore automatically are the repository and system models.

[2]An exemplary approach for generating resource profiles for Java Enterprise Edition (EE) applications is used in the evaluation section to validate this process.

[3]http://www.eclipse.org/modeling/emf/

2.4 Predicting Performance

Using PCM-based resource profiles, performance predictions can be made for different workload and hardware environment models. These predictions include results for the performance metrics response time, throughput and resource utilization. As performance change is defined as changes in the transaction processing time, response times in the prediction results are used as an indicator for change.

To enable comparable predictions in the deployment pipeline, workload and hardware environment models for performance predictions need to be statically defined. As explained in the previous section, the workload on a system can be represented using usage models. The hardware environment is represented using resource environment and allocation models. The artifact repository therefore needs to contain usage models as representation of different workloads, which use the external interfaces provided by the system model of a resource profile to specify the load. It furthermore needs to contain one or more resource environment models specifying the available servers and corresponding allocation models which define the mapping of deployment units specified in a resource profile to the servers.

Before performance can be predicted, a lookup needs to be made in the artifact repository to get the corresponding workload and hardware environment models. The workload and hardware environment models are combined with a resource profile to predict performance. If a hardware environment should be used with different hardware components than the one the resource profile was created on, the processing rates of the hardware resources need to be specified relative to the original ones. One approach to do that is to use benchmark results for the source and target hardware as shown in [6].

2.5 Comparing Prediction Results

To detect performance change, performance is predicted with the current resource profile version and a specified set of hardware environment and workload models. If a previous resource profile version for the same application is available, the same predictions are executed using the previous version. Afterwards, the prediction results are compared. The reason for executing the predictions with the previous resource profile version again is to ensure that the same workloads and hardware environments are used. This avoids situations in which changes in the workload or hardware environment models lead to incorrect results.

For each transaction, the relative predicted response time change between the current and the previous application version is calculated. As a prediction results in multiple response time values over time, this set of transaction response time values can be represented in multiple ways [10]. The most common ones are mean values including distances from the mean (e.g., standard deviation) as well as percentiles (e.g., 50th (median) or 90th percentile). Which one of these values represents a response time set best depends on the dispersion of the underlying distribution [10]. The dispersion of the underlying distribution can change between the response time sets collected for the same transaction in different versions. To account for this fact while making the results comparable between versions, the relative change is always calculated for mean and median values as well as for 90th percentiles.

We propose to calculate the relative change (rc) as shown in equation 1. In this equation, the transaction response time (rt) predictions of the *current* and *previous* resource profile versions are compared with each other. The resulting change value indicates whether the response time is now higher than before if the value is positive or lower if the value is negative.

$$rc = \frac{rt_{current} - rt_{previous}}{rt_{previous}} \qquad (1)$$

If the relative change for at least one transaction is higher than a specified threshold (e.g., above 20 % increase or decrease) a notification (e.g., as email from the build system) is being sent to the developer(s). If response time increased above a specified threshold the deployment pipeline needs to be stopped. It is important to stop the pipeline in this case, so that the application version is not marked as stable. This ensures, that the next build will be compared with a valid resource profile of a version with meaningful performance.

To enable a distinction between runs with and without changes, the corresponding resource profile versions need to be managed independently within the EMFStore and a resource profile should only be marked as usable for subsequent builds if the relative change is below the specified threshold.

To identify trends, a comparison can also be made against the resource profiles of more than one of the last successful builds. This avoids situations in which performance regressions develop slowly across multiple builds.

2.6 Comparing Resource Profiles

Users can access, analyze and edit models in the EMFStore using a plugin for the PCM modeling environment[4]. Due to this reason, resource profile versions in the artifact repository are directly accessible to developers when they are notified about a performance change. The notification could also include a link to the corresponding resource profile versions. Each resource profile in the EMFStore can be analyzed over time to see which components, relationships or resource demands are associated with specific versions stored in the repository.

As resource profiles for normal application versions and for versions with performance changes are managed independently from each other in the EMFStore, one cannot easily analyze the differences between these versions. For this purpose, one can compare the different resource profile versions using the EMF Compare framework[5]. EMF Compare allows to automatically analyze and visualize the differences between models conforming to the Ecore meta-model.

The level of available detail depends on the resource profile generation approach, but we assume that the components a system is composed of, their operations and their relationships are represented in the model [18], including corresponding resource demands. Therefore, the result of a comparison reveals changes in resource demands, control flows and component operations. Using this information, the developers can identify the sources for a performance change. Combined with the information, which changes have been performed during the check-ins that initiated the build, it should help the developer to identify the cause of a performance change.

[4]http://www.palladio-simulator.com/

[5]http://www.eclipse.org/emf/compare/

3. EVALUATION

This section evaluates the performance change detection process within a deployment pipeline as explained in section 2. Section 3.1 describes the setup of the build and test system for this evaluation. Afterwards, the evaluation steps using these systems are explained in section 3.2.

3.1 Build and Test System

The evaluation in this paper uses a SPECjEnterprise2010[6] benchmark application called Orders domain as an exemplary enterprise application. The advantage of using a benchmark application is that the benchmark specifies a workload as well a dataset for test runs so that they can be easily repeated by others. The Orders domain is a Java EE web application that is composed of Servlet, JavaServer Pages (JSP) and Enterprise JavaBean (EJB) components. 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 experiment setup consists of two virtual machines (VM). One VM, called build system, is used to execute the build and test tasks within the deployment pipeline. The other VM, called test system, is used to host the Orders domain deployment. These two virtual machines are mapped to two different hardware servers (IBM System X3755M3). The hardware servers are connected using a one gigabit-per-second network connection. Both virtual machines run openSuse 12.3 64bit as operating system and have four virtual CPU cores and 40 gigabytes of random-access memory.

The deployment pipeline on the build system is implemented as projects in the CI system Jenkins. A first project builds the Orders domain. If the build was successful, the new Orders domain version is deployed on the test system in a second project. The application is deployed on a GlassFish Application Server (AS) Open Source Edition 4.0 (build 89) in the Java EE 7.0 full profile. The database on the test system VM is an Apache Derby DB in version 10.9.1.0. The GlassFish AS and the Apache Derby DB are both executed in the same 64 bit Java OpenJDK VM (JVM version 1.7.0).

Once the deployment is completed successfully, a third project executes automated acceptance tests. For that purpose, test scripts are used that are provided by the benchmark. These

[6]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.

test scripts describe the user interactions with the Orders domain and are implemented to run within the Faban harness[7]. Faban is a workload creation and execution framework. These tests are no acceptance tests in the traditional sense, but they exercise the system in a way a normal user would and are, thus, comparable.

If the acceptance tests complete successfully, a forth project in the CI system triggers the generation of a resource profile. An automatic performance model generation approach for Java EE applications introduced in [5] is used to generate resource profiles for the Orders domain application. This approach uses runtime instrumentation to collect data for the model generation. Resource profiles are generated based on the data collected during the acceptance tests. These profiles are stored and versioned in an EMFStore server running on the build system. The last steps in the deployment pipeline shown in figure 1 are not automated.

3.2 Evaluation Steps

1.) In a first step, the automated steps of the deployment pipeline are executed for the standard version of the Orders domain application.

2.) In a second step, the Orders domain application is modified and the automated steps of the deployment pipeline are triggered again. In this version, the performance characteristics of two application components are modified by increasing their resource consumption.

3.) Before we continue to identify performance change between both application versions, the prediction accuracy of both resource profiles is evaluated. To do that, prediction results of both resource profiles are compared with measurements of their corresponding application versions deployed on the test system.

4.) To identify performance change, we are using the resource profiles of both versions. They are used to predict response times for predefined workloads and one hardware environment. The expected result is, that an increase of response time and thus a regression in performance can be observed. To identify the root cause, the resource profile versions are compared directly to see if the change introduced in the second application version is visible in this comparison.

3.3 Creating and Versioning Resource Profiles

Once the standard version of the SPECjEnterprise2010 Orders domain application is being built and deployed on the instrumented test system, acceptance tests are executed. These tests are executed with 600 concurrent users for 20 minutes while data is only collected between a five minute ramp up and a five minute ramp down phase. All of the following test runs are executed using the same duration. Once the test is completed, a command line utility is triggered by the build system that implements the approach presented in [5] and generates a resource profile for this application version and stores it in the EMFStore.

In a second step, an updated version of the Orders domain application is being built and deployed on the test system.

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

The updated version is modified so that two components of the application consume more CPU time than in their original versions. Another test run with 600 concurrent users is then executed using the updated version. Afterwards, a new version of the resource profile is generated and stored in the EMFStore.

The resource profile for the original Orders domain application is hereafter called resource profile version one and the second resource profile for the modified application is hereafter called version two. To evaluate performance for both application versions, usage and hardware environment models are predefined. The usage model is created following the source code of the test scripts in the Faban harness. The hardware environment models represent the test system.

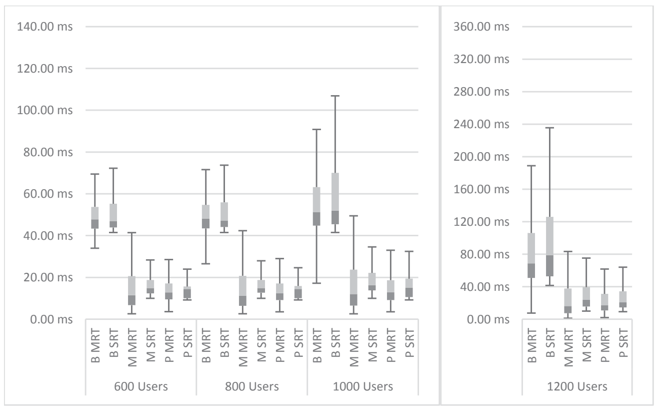
3.4 Evaluating the Accuracy of Resource Profile Predictions

To evaluate the prediction accuracy of the generated resource profile versions (V), they are used to predict the performance of the corresponding application versions under different workload conditions. The same workloads are executed as test runs using the corresponding application versions. Afterwards, the measured results are compared with the predicted results. This comparison includes the response time and throughput of the business transactions (T) browse (B), manage (M) and purchase (P) as well as the CPU utilization of the test system.

The workload for this comparison is increased in steps of 200 concurrent users (U) from 600 to 1200 (~47 % to ~90 % CPU utilization). To predict the performance of the application versions, the corresponding resource profiles including the workload and hardware environment models are used as input for the simulation engine SimuCom [2]. SimuCom performs a model-to-text transformation to generate Java code based on PCM models. This Java code is afterwards executed to start a simulation. The simulation duration is set to 20 minutes while only data between a five minute ramp up and five minute ramp down phase is used for the calculation of the simulation results.

The simulation results for the resource profile versions one and two are shown in boxplot diagrams[8] in figures 2(a), 2(b) and in table 1. Response time measurement and simulation results are described using mean and median values as well as measures 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 [10] of the response times of browse, manage and purchase. For each load condition specified by the number of users, figures 2(a) and 2(b) show the dispersion of the simulated response time (SRT) per business transaction. The simulated mean response times (SMRT),

[8]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 connected by vertical lines to form the box that indicates the 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.


(a) Application and resource profile version one

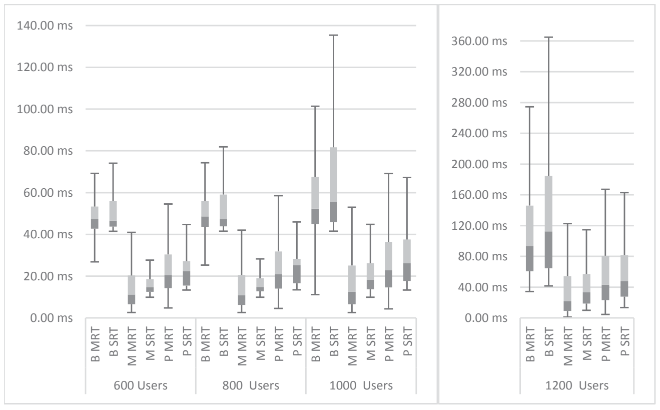

(b) Application and resource profile version two

Figure 2: Measured and simulated response times

the simulated mean CPU utilization (SMCPU) and the simulated throughput (ST) can be found in table 1.

In the same way as the simulations, the tests run 20 minutes and the data for this comparison is collected in a steady state between a five minute ramp up and a five minute ramp down phase. The measured mean CPU utilization (MMCPU) of the test system is measured during this time using the system activity reporter. Only the measured throughput (MT) values are directly taken from the Faban harness. As the simulated values do not contain the network overhead between the build and the test system, the response time values of the business transactions are calculated based on measurements performed directly on the test system. For this purpose, the response times of HTTP requests executed by the Orders domain users on the test system are measured during the test execution using a Servlet filter [5, 15]. The response times of the single HTTP requests are then used to calculate the measured response time (MRT) values for the business transactions shown in the boxplot diagrams in figures 2(a) and 2(b). The measured mean response times (MMRT) are shown in table 1.

The comparison of the simulated and measured results shows that both resource profiles do represent their corresponding application versions very well. The highest relative response time prediction error (RTPE) for the mean response time values is 15.99 % for the browse transaction and a load of 1000 concurrent users. The median values support this result as the relative prediction error for the median values is at most 21 % for the browse and purchase transactions. For the manage transaction, the relative error of the median values goes up to 52 % for a load of 1200 concurrent users in the second resource profile version.

The skewness of a business transaction's underlying response time distribution can be determined considering the median's position between the quartiles Q_1 and Q_3. The results show that the skewness of the underlying distribution is not always correctly represented. This is especially the case for manage, as the first quartile Q_1 is predicted by the simulation with a relative error of up to 112 %, which is already caused by an absolute error of 7.32 milliseconds (ms). The first quartile Q_1 for the browse and purchase transactions is mostly represented with a relative error below 22 %, only for a load of 1200 users, the relative error for the purchase transaction in the predictions with the first resource profile version goes up to 32 %. The third quartile Q_3 is predicted with a relative error of at most 26 % for all transactions.

The relative throughput prediction error (TPE) is at most 3.95 % (see table 1). This validates the results but is expected, as the think time for each user is much higher (9.8 seconds) than the response time measurement and simulation results shown in figures 2(a) and 2(b). The impact of response time prediction errors on the throughput is thus very low.

The relative CPU utilization prediction error (CPUPE) is at most 11.12 % (see table 1). The simulated CPU utilization is below the measured CPU utilization in low load levels (600 and 800 concurrent users), as the garbage collection overhead and other JVM activities are not represented in the resource profiles [5, 6]. In high load conditions (1000 and 1200 concurrent users) the simulated values are slightly higher than the measured values. The data collection overhead included in the resource demands of the model elements [5] thus seems to balance the influence of aspects not represented in the resource profiles in high load conditions.

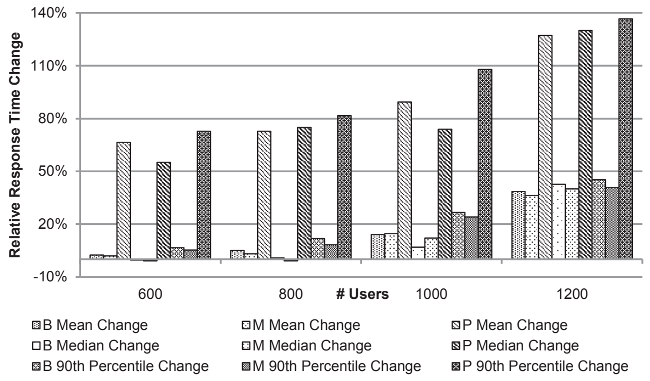
3.5 Comparing Prediction Results and Resource Profile Versions

The prediction results of the two resource profile versions are now compared with each other to identify performance change. As shown in figure 3(a), the response times of all transactions increased from version one to two, even though the median response time values of manage show a slight decrease for low load levels (600 and 800 users). It is furthermore visible that the purchase transaction response time increased in all load levels. The mean, median and 90th percentile values for this transaction increased by at least 66 % and up to 137 %. Therefore, the results show a clear regression for the purchase transaction.

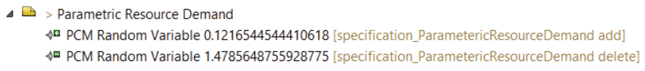
Using a plugin in the PCM modeling environment we ac-

Table 1: Measured and simulated results for resource profile versions one and two

V	U	T	MMRT	SMRT	RTPE	MT	ST	TPE	MMCPU	SMCPU	CPUPE
1	600	B	49.95 ms	53.13 ms	6.38%	18,027	18,328	1.67%	47.48 %	43.12 %	9.18 %
		M	17.90 ms	16.51 ms	7.72%	8,970	8,989	0.21%			
		P	14.03 ms	14.53 ms	3.56%	8,946	9,275	3.68%			
	800	B	51.82 ms	54.05 ms	4.31%	24,332	24,524	0.79%	59.81 %	57.60 %	3.70 %
		M	18.28 ms	16.93 ms	7.39%	11,912	12,066	1.29%			
		P	14.25 ms	14.76 ms	3.60%	11,972	12,194	1.85%			
	1000	B	59.13 ms	62.91 ms	6.40%	30,274	30,454	0.68%	71.26 %	71.76 %	0.70 %
		M	21.66 ms	19.64 ms	9.33%	15,081	15,281	1.33%			
		P	16.19 ms	17.05 ms	5.31%	15,122	15,252	0.86%			
	1200	B	88.57 ms	98.78 ms	11.53%	36,389	36,497	0.30%	81.17 %	85.97 %	5.92 %
		M	35.45 ms	30.92 ms	12.79%	17,836	18,540	3.95%			
		P	27.38 ms	26.97 ms	1.51%	18,119	17,975	0.79%			
2	600	B	49.50 ms	54.45 ms	9.99%	18,196	18,442	1.35%	51.65 %	45.90 %	11.12 %
		M	17.80 ms	16.83 ms	5.45%	9,078	9,124	0.51%			
		P	24.48 ms	24.20 ms	1.15%	9,146	9,016	1.42%			
	800	B	52.69 ms	56.80 ms	7.81%	24,308	24,332	0.10%	63.51 %	61.01 %	3.93 %
		M	18.41 ms	17.46 ms	5.15%	12,118	12,165	0.39%			
		P	25.76 ms	25.49 ms	1.05%	12,034	12,292	2.14%			
	1000	B	61.85 ms	71.74 ms	15.99%	30,379	30,326	0.17%	73.98 %	76.05 %	2.81 %
		M	23.71 ms	22.49 ms	5.14%	14,880	15,406	3.53%			
		P	29.74 ms	32.31 ms	8.62%	15,021	15,186	1.10%			
	1200	B	120.88 ms	136.82 ms	13.19%	36,318	36,394	0.21%	86.68 %	90.98 %	4.97 %
		M	49.18 ms	42.16 ms	14.27%	18,010	18,106	0.53%			
		P	63.24 ms	61.24 ms	3.15%	18,271	18,273	0.01%			



(a) Relative change between both resource profile versions



> Parametric Resource Demand
PCM Random Variable 0.1216544544410618 [specification_ParametericResourceDemand add]
PCM Random Variable 1.4785648755928775 [specification_ParametericResourceDemand delete]

(b) EMF Compare result

Figure 3: Comparison results

cessed the EMFStore that contains both resource profile versions and looked at their differences using EMF Compare. A screenshot of one comparison result can be found in figure 3(b). This screenshot shows, that the CPU resource demand of one component operation (1.48 ms) is about twelve times higher than the one of the previous version (0.12 ms). A similar result is visible in two components involved in the control flow of requests within the purchase transaction. Comparing the resource profiles, thus, helped to identify the places which caused the performance regression.

4. RELATED WORK

Mi et al. [14] and Cherkasova et al. [7] have already identified the need to evaluate the performance changes in each enterprise application version. For this purpose, the authors propose to create so called application signatures and compare the performance characteristics of each application version using their corresponding signatures. Application signatures are a representation of transaction processing times relative to the resource utilization for a specific workload. The authors extend their approach in [8] by using performance modeling techniques to not only detect performance changes based on response times but also to evaluate performance anomalies in the CPU demand of transactions. Their work is focused on systems that are already in production. This work extends the idea of evaluating the change in performance of each application version to the development process. Furthermore, we propose the use of resource profiles instead of application signatures. Resource profiles allow for more flexible evaluations as they can be used to derive these metrics for different workloads and hardware environments. It is thus possible to derive more meaningful metrics with smaller systems as it would be possible using application signatures. This is important for performance evaluations during software development, as performance test environments might not be available [4].

An approach to detect performance regressions of different versions of a software system during development has been proposed by Nguyen et al. [16]. The authors propose the use of control charts to identify whether results of a performance test indicate a regression or not. Their approach focuses on the automatic detection of regressions based on performance test results. It could be used to evaluate prediction results based on resource profiles. However, their approach requires a real performance test and does not support the detection of possible problem causes as is possible by comparing resource profiles. It furthermore assumes a linear relationship between the resulting performance metrics and the load on a system, which might not always be true.

5. CONCLUSION AND FUTURE WORK

Detecting performance change as part of a deployment pipeline provides immediate feedback to developers about the impact of a change on the performance of an overall enterprise application. The performance metrics and their relative change values indicate whether an application is on track to achieve the required performance goals. Compared to a performance evaluation solely at the end of the software development process, the suggested approach allows tracking and improving the performance along with the functionality in the process.

The evaluation results validate the performance change detection process within a deployment pipeline for a Java EE application. However, there are some limitations of the approach that need to be dealt with in future work. As of today, performance change can only be detected when the resource demand type which causes a performance change is also represented in a resource profile. As representing memory is currently not fully supported by the underlying meta-model, changes caused by different memory consumption characteristics are not detectable yet.

An additional open challenge is that external interfaces of an application can change between enterprise application versions. If one resource profile version breaks the compatibility with the interfaces used by the workload specifications in the artifact repository, the change detection will stop working as long as the usage models are not modified. Even if a corresponding modification is made, the new usage models will no longer be compatible with the previous versions.

Another direction of future work is to automate the complete performance change detection process and to integrate it as a plugin in CI/CD systems. Furthermore, the steps of comparing different resource profile versions and identifying the problem causes should be handled automatically. It might also be interesting to use information about check-ins that are included in a build to perform static analysis to improve the search process for the reasons of a change.

6. REFERENCES

[1] S. Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Karlsruhe Series on Software Quality. Universitätsverlag Karlsruhe, 2008.

[2] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[3] R. Brandl, M. Bichler, and M. Ströbel. Cost accounting for shared it infrastructures. *WIRTSCHAFTSINFORMATIK*, 49(2):83–94, 2007.

[4] A. Brunnert, C. Vögele, A. Danciu, M. Pfaff, M. Mayer, and H. Krcmar. Performance management work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[5] A. Brunnert, C. Vögele, and H. Krcmar. Automatic performance model generation for java enterprise edition (ee) applications. In M. S. Balsamo, W. J. Knottenbelt, and A. Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2013.

[6] A. Brunnert, K. Wischer, and H. 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.

[7] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, DSN '08, pages 452–461, 2008.

[8] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Automated anomaly detection and performance modeling of enterprise applications. *ACM Transactions on Computer Systems*, 27(3):6:1–6:32, Nov. 2009.

[9] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[10] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[11] B. King. *Performance Assurance for IT Systems*. Taylor & Francis, 2004.

[12] M. Koegel and J. Helming. Emfstore: A model repository for emf models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 307–308, New York, NY, USA, 2010. ACM.

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

[14] N. Mi, L. Cherkasova, K. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *IEEE Network Operations and Management Symposium*, NOMS '08, pages 216–223, 2008.

[15] R. Mordani. Java servlet specification v2.5, 2007.

[16] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[17] S. Spinner, G. Casale, X. Zhu, and S. Kounev. Librede: A library for resource demand estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 227–228, New York, NY, USA, 2014. ACM.

[18] X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Software Engineering Notes*, 29(1):290–301, 2004.