Systematic Performance Evaluation based on Tailored Benchmark Applications

Christian Weiss
SAP Research
Vincenz-Priessnitz-Strasse 1
76131 Karlsruhe, Germany
christian.florian.weiss@sap.com

Dennis Westermann
SAP Research
Vincenz-Priessnitz-Strasse 1
76131 Karlsruhe, Germany
dennis.westermann@sap.com

Christoph Heger
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
christoph.heger@kit.edu

Martin Moser
SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
martin.moser@sap.com

ABSTRACT

Performance (i.e., response time, throughput, resource consumption) is a key quality metric of today's applications as it heavily affects customer satisfaction. SAP strives to identify and fix performance problems before customers face them. Therefore, performance engineering methods are applied in all stages of the software lifecycle. However, especially in the development phase continuous performance evaluations can introduce a lot of overhead for developers which hinders their broad application in practice. In order to evaluate the performance of a certain software artefact (e.g. comparing two design alternatives), a developer has to run measurements that are tailored to the software artefact under test. The use of standard benchmarks would create less overhead, but the information gain is often not sufficient to answer the specific questions of developers. In this industrial paper, we present an approach that enables exhaustive, tailored performance testing with minimal effort for developers. The approach allows to define benchmark applications through a domain-specific model and realizes the transformation of those models to benchmark applications via a generic Benchmark Framework. The application of the approach in the context of the SAP Netweaver Cloud development environment demonstrated that we can efficiently identify performance problems that would not have been detected by our existing performance test infrastructure.

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [measurement techniques, design studies, performance attributes]; D.2.5 [**Software Engineering**]: Testing and debugging—*testing tools*; D.2.5 [**Database Management**]: Logical Designs—*data models*

General Terms

Java Persistence API (JPA), Benchmarking, Measurelet, Automated Performance Testing, Model-based Testing, Performance Regression Testing

Keywords

Performance, Benchmarking, Database, JPA, Feedback

1. INTRODUCTION

Identifying performance problems before customers face them is highly relevant for web-based software solutions such as websites or Software- and Platform-as-a-Service (SaaS/-PaaS) products. Recent industrial studies and field reports have shown that performance (response time, throughput, resource usage) often directly correlates with revenue [16]. Engineering methods (e.g., monitoring, testing, modelling) can help software providers to understand performance behaviour and identify performance problems early.

However, evaluating performance in a realistic environment involving large subsystems (e.g., databases, middleware, legacy) is still a challenging task. Especially in the development phase, continuous performance evaluations can introduce a lot of overhead for developers which hinders their broad application in practice.

In this industrial paper, we report on our efforts towards a systematic and lean approach for evaluating the performance of the persistence layer of a platform solution. The persistence layer of an application is often one of the most performance-critical components as it is quite likely that it becomes a bottleneck. In a PaaS solution, platform providers offer a ready-to-use persistence layer to developers that create applications on top of the platform. This implies the following challenges.

The platform service provider needs to support many different application types with different usage profiles. Standard benchmarks such as provided by SPEC [15] and TPC [18] do not test broad enough in order to enable a detailed understanding of performance characteristics. In order to simulate a comprehensive set of workloads at development time, it requires tailored performance tests that simulate many different usage patterns and specifically focus on the concrete system under test [7]. However, creating and running these detailed tests is very time-consuming and requires detailed expert knowledge.

Developers that consume the platform service are mainly interested in understanding how the performance of the platform service affects the performance of their application. The platform service provider could support the consumers by providing a prediction that maps the service usage to the expected performance. However, in order to provide such a prediction, the platform service provider needs an even more detailed understanding of how the service behaves under different usage profiles. In our scenario, classical model-driven prediction approaches (such as surveyed in [10]) fail due to the complexity of the service and its continuous modification which makes it hard to create and maintain the models. Hence, exhaustive, systematic measurements are the means that we choose to derive predictions. This underlines our requirement for an efficient approach to set up and run tailored performance tests.

In this paper, we present our approach for creating and running performance tests tailored to the persistence service of a platform solution. In our scenario, the service interface is the Java Persistence API (JPA) [3]. Based on a JPA data model provided by the platform consumer, an object-relational mapping (ORM) component persists the application data in the underlying database. We introduce a measurement API that separates benchmark applications from their runtime environment. Moreover, we developed a Benchmark Framework that generates all necessary project and source files of a benchmark application based on information provided in a benchmark model. Furthermore, the Benchmark Framework compiles the generated code, deploys it on the test environment, executes the measurements and returns the results.

The presented approach simplifies the creation and execution of performance tests for developers by hiding complexity and automating many tasks. This simplification makes it easier to introduce performance tests which results in a larger set of tests. Having more tailored tests increases the probability of uncovering a performance problem. Moreover, it allows us to quickly reproduce a performance issue on the test environment that we observed for a consumer application on the real environment. In the paper, we demonstrate how we use the approach for performance regression testing and performance prediction function construction. Moreover, we report on the first results that we achieved by applying the approach in the development of the SAP Netweaver Cloud platform [14]. Based on the initial experience with the approach, we discuss open issues and future research directions that we see in the performance engineering field.

The remainder of this paper is organized as follows. Section 2 gives an overview on the approach and introduces the key components. Sections 3 and 4 describe the measurement API and our approach to automate the construction, execution and analysis of benchmark applications, respectively. In Section 5, we demonstrate the application of the approach in our industrial setting. In Section 6, we discuss potential research directions based on our experience with the presented approach. Related research is introduced in Section 7. Finally, Section 8 concludes the paper.

2. OVERVIEW

The main goal of the work presented in this paper is to support developers as well as consumers of a persistence service in understanding the performance impact of design and implementation decisions. Figure 1 illustrates the basic scenario.

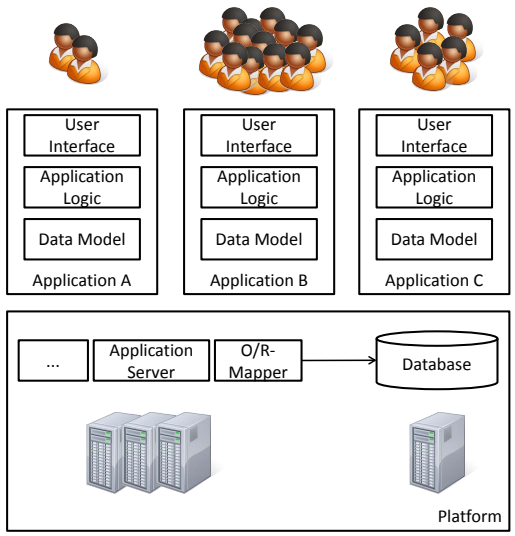

Figure 1: Platform Scenario

The platform hosts a set of different applications with different usage profiles. In addition to hosting the applications, the platform provides further services to the application developers. In this paper, we focus on the persistence service of the platform which allows application developers to store and load data based on a simple interface. Hence, application developers do not have to deal with setting up, configuring, or maintaining their own persistence solution. The results presented in this paper are derived for a persistence service that uses JPA [3] as the interface to the application developer. In that scenario, the main components of the persistence service are the database and the object-relational mapping component (such as Hibernate [6] or EclipseLink [23]). The application developer designs the entities that make up the data model of the application and bundles it together with the application logic and the user interface to a deployable unit that runs on the the platform.

When it comes to quality assurance in terms of performance, the persistence service provider as well as the application developers need to answer different questions. Examples from a persistence provider view are:

- Does my planned database update affect performance?
- Which test cases reflect the actual usage profile best?
- Should I use O/R-Mapper A or B?

Application developers face questions like:

- Is variant A or variant B of my data model better with respect to performance?
- What is the performance of my application on platform X?
- Did my latest change affect the performance of my application?

The approach presented in this paper provides a means to support persistence service providers and application developers in answering these questions efficiently. In Section 5, we provide some examples on how we used the approach for performance regression testing and providing performance feedback. Figure 2 illustrates the basic idea of the approach based on a simple example.

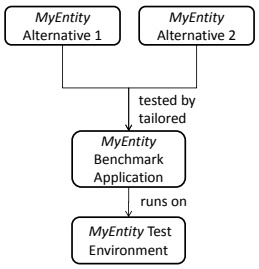

Figure 2: Basic Idea

If the developer of a data model wants to make a decision between two design alternatives for the entity *MyEntity*, a tailored benchmark application is constructed that tests the performance of the two alternatives of this specific entity on a test environment. Thereby, the design of the benchmark application as well as the test environment depend on the goal of the test (e.g. comparing two design alternatives with respect to an achievable throughput for a certain workload). This procedure requires to create and run a large set of tailored measurements. In the following, we give an overview on how we manage this requirement.

Creating and running a single performance measurement in the described context requires the following steps:

1. Developing a JPA data model.
2. Developing a test application logic that uses the data model.
3. Developing a load driver that simulates a usage profile.
4. Adding instrumentation that enables performance monitoring.
5. Deploying the test application on a performance test environment.
6. Executing the performance test.
7. Analysing and interpreting the measured data.

In the following, we refer to the software that is created during steps 1 to 4 as benchmark application. With our approach, we aim at reducing the manual effort in creating and executing benchmark applications while retaining the flexibility in creating these individual benchmark applications. Therefore, we facilitate the definition of such benchmark applications through a generic domain-specific model and provide a Benchmark Framework to realize the transformation of those models to benchmark applications (see Section 4) which allows us to efficiently create and run benchmark applications. Furthermore, we adopt a concept known from the web development domain where developers build *Servlets* that are executed in a *Servlet Container* [12]. The *Servlet Container* provides the infrastructure to run the *Servlets* and takes over management and controlling of *Servlets*. In our approach, the benchmark applications are called *Measurelets* and the mechanisms to control such *Measurelets* are provided by a *Measurelet Container*. A *Measurelet Container* executes *Measurelets* by providing an appropriate infrastructure and can provide additional services concerning the processing of measurements (e.g., recovery of failed runs or repeating runs until a certain confidence level has been reached). The benefit of this approach is that the developer of a benchmark application can focus on the design of the benchmark application (i.e., the *Measurelet*) and does not have to deal with setting up the execution environment and controlling the measurements. In order to ensure the compatibility between the *Measurelets* and the *Measurelet Container* we introduce a *Measurelet API* as part of the approach (see Section 3). The *Measurelet API* supports developers in building and running a single benchmark application. However, in order to answer some of the questions stated above, the construction and execution of a multitude of benchmark applications is required (e.g. for exhaustive performance regression testing such as described in Section 5).

Developing all these benchmark applications manually would be too time-consuming and thus impracticable. Therefore, we introduce a Benchmark Framework that allows us to automatically create, run and analyse *Measurelets*. Thereby, the construction and execution of benchmark applications can be parametrisable. A *Measurelet* can, for example, contain the code of the application logic that accesses the data model via a certain interface. The actual implementation of the data model can then be varied which allows comparing different design alternatives. In order to support the efficient parametrised construction and execution of a multitude of *Measurelets*, we introduce a model-based representation of *Measurelets*, called JPA Benchmark Model (JBM). Having the information about the content of a *Measurelet* as well as a set of variation points specified in a model allows us to leverage model-driven development advantages such as code generation and consistent design documentation. The model is interpreted by the Benchmark Framework that automates the execution of the three steps illustrated in Figure 3.

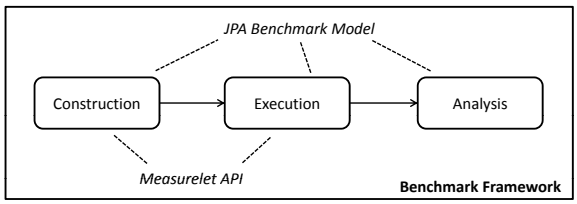

Figure 3: Benchmark Framework

The Benchmark Framework automates construction, execution, and analysis of benchmark applications. It is parametrisable via the information specified in the JPA Bench-

mark Model. In the construction step the Benchmark Model triggers the generation of code and configuration files and packages the benchmark application (i.e., the *Measurelet*) to a deployable unit. In the execution phase the *Measurelet* is deployed to the *Measurelet Container* which runs the application. Finally, the analysis step analyses the measured results and correlates the measurements with the information specified in the JPA Benchmark Model. In Section 4, we introduce this process in more detail.

Both, the *Measurelet* concept as well as the automation capabilities of the Benchmark Framework combined with the JPA Benchmark Model helped us to reduce the effort for detailed performance testing. In Section 5, we report on the first results that we achieved by applying exhaustive, tailored performance measurements in the development process of a platform persistence service.

3. MEASURELET API

The *Measurelet API* is an approach to reduce the efforts for creating and executing tailored performance measurements. We apply the idea of the *Servlet API* [12] to the performance measurement domain. Benchmark applications are called *Measurelets* and created based on a common abstract interface. The execution of Measurelets is controlled by so-called *Measurelet Containers*. Hence, a Measurelet is focused on benchmark structure, workloads, and monitoring probes. The management and execution of Measurelets is conducted by Measurelet Containers based on the abstract Measurelet API. The Measurelet Containers adapt to the specific platform on which the Measurelets (i.e., the benchmark applications) should be executed (i.e. the SAP Netweaver Cloud platform in our scenario).

Figure 4 shows the common interface for the implementation of Measurelets.

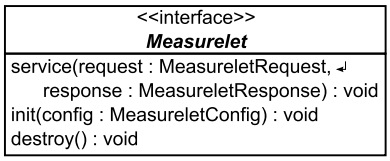

Figure 4: Measurelet Interface

The method `service` realizes the actual measurement. This method expects two input parameters which represent the interface to the caller of the benchmark application. The parameters provide access to the caller input and return the response values. The values for those parameters are provided by the Measurelet Container, which abstract, for example, from the concrete communication protocol or deployment format of the execution platform. Examples for different communication protocols are HTTP and Java Remote Method Invocation (RMI) for synchronous invocations and messaging protocols like the Advanced Message Queuing Protocol (AMQP) for asynchronous invocations.

To create a Measurelet, a developer has to create a class which implements the Measurelet Interface shown in Figure 4. The developer does not need to provide extensive, platform-specific execution and communication logic as this is realized in the Measurelet Containers. Rather, he can focus on the core benchmark application design and rely on the common interfaces to access the inputs and provide the outputs. Using a common, abstract interface for the implementation of Measurelets and their execution through Measurelet Containers makes the implementation of these components independent from each other. Thus, Measurelets are not bound to a specific Measurelet Container, which makes them executable on different platforms which cover, for example, different JPA-based persistence services. Listing 1 shows an example Measurelet implementation.

```
1  public class MyMeasurelet
2      implements Measurelet {
3    public void service(
4        MeasureletRequest request,
5        MeasureletResponse response) {
6      int count = request.getParam("count");
7      long time = System.nanoTime();
8      for (int i = 0; i < count; ++i) {
9        persist(createRandomEntity());
10     }
11     time = System.nanoTime() - time;
12     response.setValue("persist.time", time);
13   }
14   // ...
15 }
```

Listing 1: Sample Measurelet Implementation

The benchmark application is realized within the `service` method and uses the input parameters provided by the caller to control a measurement and evaluate its result. Line 6 shows how input parameters are accessed. In line 12 the measured value is stored in the result object. The implementation artefacts of such a Measurelet code can be deployed and executed on Measurelet Containers using its integration tools. For example, such a tool could package the artefacts in a JAR file to deploy and invoke it on a remote application server.

In the following sections, we present an approach that aims at the automated construction and execution of Measurelets using a model-based description. This allows us to create and run an exhaustive set of benchmark applications for a certain target platform.

4. AUTOMATED, TAILORED BENCHMARKING

Exhaustive, tailored benchmarking can be used in multiple scenarios (see also Section 5). Examples are (1) performance regression testing, (2) comparing performance characteristics of different platforms, or (3) comparing design alternatives. In all use cases, the following basic actions are to be conducted (cf. Figure 3):

- create a benchmark application,
- execute the benchmark application on the target platform, and
- analyse the results.

While all use cases have to execute the same actions, they differ in how the actions are included in a process flow. For example, in the performance regression testing use case, the same benchmark application is executed multiple times while the execution platform evolves. Whereas, in the design alternative comparison use case, we compare the results of two different benchmark applications that are executed on the same platform.

The effort to implement the use cases mentioned above can be very high, especially if the basic actions have to be implemented manually for each use case. Hence, our approach extracts the common actions into components that can be reused in different use cases. This allows us to simply compose the basic actions in order to implement the process flow of a specific use case. Moreover, we use a model-based description of benchmark applications that allows us to leverage advantages of model-driven development such as code generation and consistent design documentation.

In the following sections, we introduce our JPA Benchmark Model as well as the component-based Benchmark Framework that we developed to automate the creation, execution and analysis of benchmark applications.

4.1 JPA Benchmark Model

A *JPA Benchmark Model* (JBM) is a means to describe benchmark applications that are supposed to run on a platform that provides a JPA-based persistence service. Using a model, performance analysts can provide a declarative description of benchmark applications. This description is then used to automatically construct, execute and analyse the benchmark application (see Section 4). A JPA Benchmark Model is constructed based on a metamodel that allows a complete specification of JPA entity structures as it is aligned to the elements of the *Metamodel API* described in the JPA Specification [3]. Moreover, the metamodel allows for the registration of additional information in a key-value fashion. This additional information is required to describe the other parts (i.e., other than the data model) of the benchmark application (e.g. information about test data or test workload). Listing 2 illustrates an example for the information described in a JPA Benchmark Model.

```
1  JPABenchmarkModel:
2      entities:
3          - name: Customer
4            operationsToTest: [insert, update]
5            attributes:
6                - name: id
7                  type: Integer
8                  id: true
9                - name: primaryAddress
10                 type: String
11                 length: 128
12               - name: secondaryAddresses
13                 type: List<String>
14                 size: 4
15                 stringItemLength: 128
```

Listing 2: Custom JPA Benchmark Model

The model consists of a single entity class called `Customer`, which has 3 persistent attributes called `id`, `primaryAddress` and `secondaryAddresses`. Line 4 represents the additional information which states that for this entity class the performance of the persistence operations `insert` and `update` should be tested. In line 11, additional information is added to the attribute `primaryAddress` which defines to use string values with a length of 128 characters. Lines 14 and 15 specify that the list of `secondaryAddresses` should contain 4 items and the length of those string items should be 128 characters.

A JPA Benchmark Model can be used to automatically create executable applications through model to code transformation which has several benefits. Experts like benchmark designers can create the necessary transformations whereas application developers and performance analysts create the model. Moreover, it is possible to create model instances via an API which allows to combine the approach with different applications and tools such as systematic experimentation [20] or code extraction [5]. In the following section, we provide a detailed description on how the models are used to automate the benchmarking process.

4.2 Benchmark Framework

We developed a *Benchmark Framework (BF)* which implements this component-based approach by providing a generic component model for the three basic actions outlined above. For each action, we define a specific component type. The *BF* manages the components and enables their usage in multiple use cases. The generic component model of the framework is based on OSGi [17]. Components are created and managed as OSGi services and are available through the service registry. Each action type specifies the interface of the implementing service component leading to the specification of an interface for constructing, executing, and analysing benchmark applications. The interfaces form the basis for component implementations and ensure their type-safe usage. In the following subsections, we describe the different component types and their interfaces in more detail. Moreover, we describe how we implemented the components to apply them in the use cases introduced in Section 5.

4.2.1 Construction Component Type

The construction component is responsible for creating benchmark applications. Therefore, it uses the information specified in a *JPA Benchmark Model (JBM)* to build an executable application via a model-to-code transformation. The resulting benchmark application realises the measurements of interest. The construction component accepts an instance of the *JBM* as input and provides an executable JAR file including the benchmark application as output (see Figure 5).

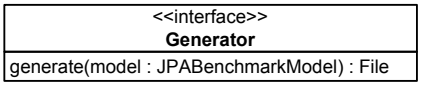

Figure 5: Construction Component Interface

The benchmark applications are implemented as *Measurelets* in order to increase the reusability of the component (see also Section 3). The construction component applied by the use cases described in Section 5 uses Apache Velocity Templates [1] to generate Java source code. The place holders and control structure within the templates are aligned to the elements of the *JBM* which is why an JBM is the only input parameter required by the templates. Listing 3 outlines an example Velocity template used to create the source code of an entity class.

```
1  @Entity
2  public class ${entity.name} {
3    #foreach ( $att in $entity.attributes )
4      #if ( $att.id == true )
5        @Id
6      #end
7      private ${att.type} ${att.name};
8      // ...
9    #end
10   // ...
```

```
11    public static class Factory {
12      public ${entity.name} randomInstance() {
13        ${entity.name} ri =
14           new ${entity.name}();
15        #foreach ($att in $entity.attributes)
16          #if ( $att.type == "String" )
17            ri.${att.name} =
18                randomString(${att.length});
19          #end
20        #end
21        return ri;
22    }}}
```

Listing 3: Sample Code Template

Lines 3-9 iterate over the attributes of an entity defined in the *JBM* and create the attribute signatures in the entity source code. In line 15, the attribute information defined in the JBM is used to generate the code for the factory method that is responsible for creating instances of the entity. Here, the template also considers additional information like the length of a String attribute (cf. line 18 in Listing 3 and line 11 in Listing 2).

The implementation uses the visitor pattern to create the different artefacts specified in the JBM. The visitor traverses the model and calls the template engine appropriately. The artefacts are created within a folder according to the Java conventions. The artefacts include the source code of JPA entities, the JPA persistence configuration file, as well as the Measurelet interface and the corresponding configuration as an OSGi declarative service. The resulting artefacts are compiled by the Java Compiler API and bundled in a JAR file using the JVM libraries of the `java.util.zip` package.

4.2.2 Execution Component Type

The execution component is responsible for running the benchmark applications on the target platform and gathering the metrics of interest. The execution can be influenced by parameters specified in the *JBM* (e.g. by specifying the database instance that should be used for the measurements). The input of the execution component is (i) a JAR file that contains an executable benchmark application and (ii) the corresponding *JBM* (see Figure 6). The output are the results of the execution (e.g. measured response times for a certain service call).

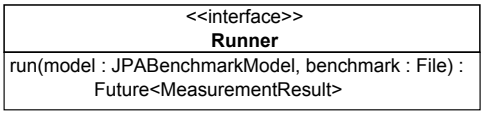

Figure 6: Execution Component Interface

The execution is based on the Measurelet API which increases the reusability of the component. The execution component used by the use cases described in Section 5 implements a Measurelet Container that adapts to the *SAP Netweaver Cloud* platform. The container registers a Servlet according to the *OSGi Web Application Specification* [17]. The Servlet allows to load JAR files into the container that are then installed as OSGi bundles. The bundles can register Measurelets as OSGi services and registered Measurelets can be executed via the Servlet. Thus, the deployed benchmark applications can be triggered by other applications using the HTTP protocol (e.g. by a nightly-build job for performance regression testing).

4.2.3 Analysis Component Type

The results derived by running one or many benchmark applications are processed by analysis components. Analysis components can perform a variety of analysis and data processing methods. For example, results can be illustrated in graphs or processed for textual representation. Moreover, it is possible to trigger follow-up actions based on the measurement results. When analysing the measured data, the component implementation can use the information specified in the *JBM*. Hence, the input of the analysis component is the execution result of a benchmark application and the corresponding *JBM* describing the application and the execution context (see Figure 7).

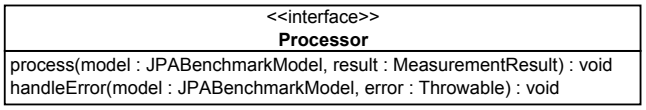

Figure 7: Analysis Component Interface

Despite the fact that the interface only accepts the result of a single execution of a benchmark application as input, a complementary component can enhance the capabilities by storing and providing results of previous benchmark application runs. This allows to implement more sophisticated analyses such as the performance regression analysis applied in one of the use cases described in Section 5.

5. USE CASES

In this section, we introduce two use cases that leverage from the approach presented in this paper. In the performance regression testing use case, we apply the approach to run an exhaustive set of tailored performance test for the persistence service of the SAP Netweaver Coud platform. The test are triggered by a nightly job on the build server and thus ensure that the development teams are timely informed on introduced performance problems. The second use case describes how we use the approach to provide performance feedback in the development environment of application developers that use the persistence service within an application that runs on the Netweaver Cloud platform.

5.1 Performance Regression Testing

Performance regression testing is a commonly used means to monitor whether a new software version still adheres to a specified quality goal or agreement. Performance tests are (in most cases) automatically executed on a regular basis (e.g., nightly). If a performance regression is observed, the development team can immediately dig into the issue. However, the probability of actually observing an issue as well as the effort for identifying its root cause is highly dependent on the number and quality of performance tests executed on a regular basis. Using the approach presented in this paper, we have been able to increase the number of performance regression test significantly. Moreover, it allows us to create performance tests that are tailored to the specifics of the persistence service which improves the value of the information provided by the tests. We developed a set of benchmark applications that include different data models, different application logic methods accessing a data model as well as different types of usage profiles. As a result, we

have an exhaustive set of benchmark applications all testing a different aspect of the persistence service. This set of benchmark applications is executed on a nightly basis. Moreover, we can easily add additional tests as soon as we identify a performance problem in an application that has not been detected by the existing tests. The concept is similar to the unit testing approach where specific tests are created for each concrete feature and tests are added after fixing a problem in order to prevent from introducing the problem again. Figure 8 shows a performance regression that we observed after having the tailored benchmark set in place.

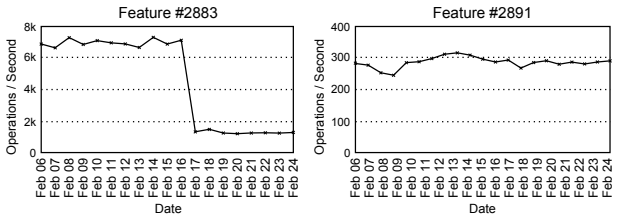

Figure 8: Identified Regression

The graphs show the measured throughput for two benchmark applications over a certain period of time. The benchmark application on the left side executes a named query that retrieves all instances of an entity in a certain data model. The benchmark application on the right side executes a query that stores a number of instances of the same entity to the database. The graph on the left side of Figure 8 shows that for this benchmark application a performance regression of factor 4 has been introduced. As the tests run on a nightly basis, we have been able to identify the root cause for the issue very quickly which happened to be an update of the database version that has been conducted at that day. An interesting observation is that the regression has only been observed in one test out of the set of benchmark applications. The test shown on the right side of Figure 8 does, for example, not show a performance regression. This observation underlines the assumption that more and tailored performance tests increase the probability of detecting a performance issue. Moreover, knowing the exact conditions under which a problem occurs and under which not can be very helpful in fixing a performance issue.

5.2 Performance Feedback

In data-centric applications, the data model can limit performance and scalability of the overall application. When developing the data model and the application logic that uses the data model, it is often unclear to developers how different design decisions or usage profiles affect the application's performance. With our ongoing research [19], we aim at increasing the performance awareness of developers by providing immediate feedback about the expected performance of the artefact under development. We integrate the performance feedback into the development environment (e.g. eclipse) to lower the burden for developers (cf. Figure 9). For the persistence scenario described in this paper, we identified two feedback usage types.

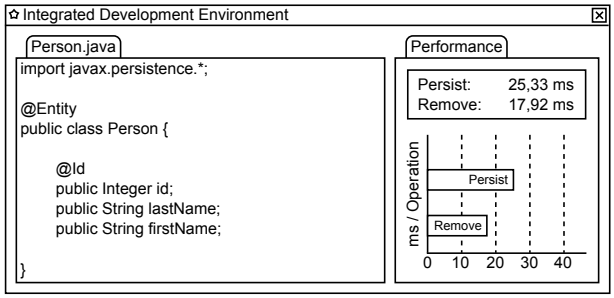

Figure 9: Immediate Feedback in the IDE.

1. Developers can use the feedback to continuously track the performance impact of changes applied to the artefact.

2. Developers can use the feedback to evaluate design alternatives with respect to data model entities (e.g. distribution of attributes across entity classes) and entity usage (e.g. number of parallel reads).

For the first usage type, the performance feedback relates to the currently focused software artefact (e.g. a data model entity or a method using persistence operations) and the performance values are updated when changes are applied to the software artefact. Figure 9 sketches how this could look like in the IDE of the developer.

While the developer is developing the entity *Person* and adding additional attributes, the performance feedback view on the right side of the figure shows how the changes affect the average response time for persist and remove operations on this entity for a predefined test workload.

In the second scenario, developers directly compare different implementation alternatives against each other in order to understand the performance characteristics of each alternative. Figure 10 shows an example for this kind of feedback.

Part (a) of Figure 10 sketches the two implementation alternatives. The functional requirement for the developer is to store 32 numbers in a *Container* entity. *Alternative 1* implements this requirement by adding 32 fields of type *Long* to the entity. *Alternative 2* uses a list field that can hold values of type Long. Part (b) of Figure 10 illustrates the performance feedback view for that example. It shows the throughput that can be achieved for the insert, update, remove, and persist operations using the respective alternative. In the example, the throughput that can be achieved with *Alternative 1* is 4 times higher than with *Alternative 2*. We derived the feedback for this example by integrating the two entities in two benchmark applications that differ only in the described point. The workload as well as the test platform have been identical.

Basically, the feedback can be based on two sources. Either by directly measuring the performance of the artefact of interest (as we did it for the example above) or by predicting the performance using performance prediction models. Measured performance is more accurate but is not immediately available to developers. Performance prediction is immediately available but requires the prior derivation of performance prediction models. In the remainder of this section, we describe how we apply the approach presented in this paper to implement the two feedback sources.

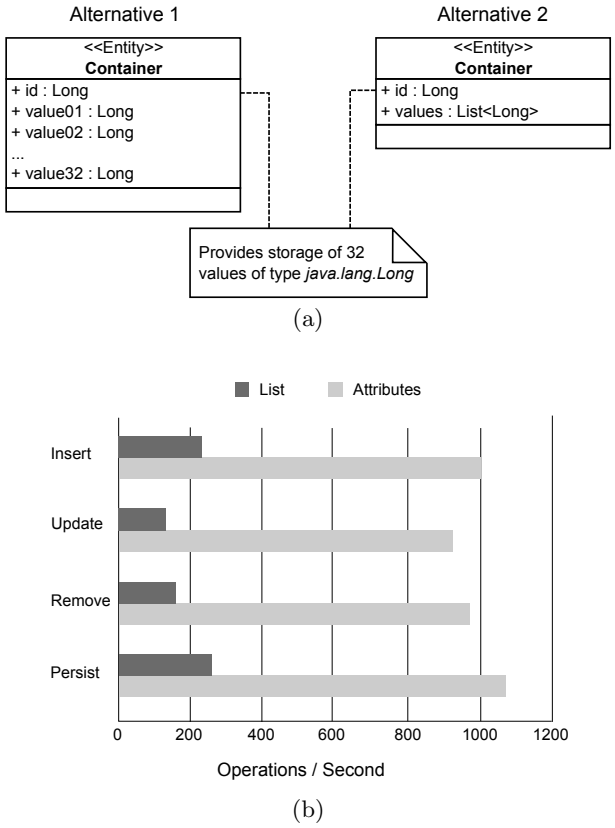

Figure 10: Evaluation of Design Alternatives.

5.2.1 Measurement-based Feedback

In case of measurement-based feedback, we execute a benchmark application on a test environment and measure the performance of different persistence service operations. Therefore, a JPA Benchmark Model is created by the IDE plugin. The plugin uses static code analysis to derive information about the data model that is to be tested and adds information about the test data and test workload that is to be used for the test. Then, the Benchmark Framework is applied to create a benchmark application based on the model, run it on the test platform and return the measurement result to the IDE plugin which channels the feedback to the developer.

5.2.2 Prediction-based Feedback

For the prediction scenario, we do not have real-world results yet. However, we give a short outline on the overall idea in order to illustrate how the approach presented in this paper can be used for our ongoing research. Our approach to derive performance prediction models is based on systematic experimentation by applying measurements and sophisticated statistical analyses (see also [21]). To define and control the experiments required to derive a prediction function, we apply our Software Performance Cockpit tool [20] in combination with the approach presented in this paper. The Software Performance Cockpit executes experiments by systematic variation of parameter values (e.g. the number of the entity's Long attributes). The benchmark framework creates and executes benchmark applications with the parameters specified by the Software Performance Cockpit and returns the performance measurement results. Based on the results provided by multiple benchmark applications SoPeCo derives a functional relationship between a set of input parameters and a performance metric of interest. For the example shown in Figure 10, the function that evaluates the achievable throughput for the persist operation could be as follows:

$$TP_{persist}(n_{LongAtt}) = 2300 + (-12 * numLongFields)$$

Deriving a complete prediction function for different data models and applications is a challenging task and requires detailed future research (see also Section 6). However, the approach presented in this paper forms the basis for this research as it provides an efficient means to run an exhaustive set of benchmark applications. Our next steps towards the automated construction of performance prediction functions for JPA-based applications are setting up systematic experiments to identify performance-relevant aspects of JPA-based data models and to identify a set of synthetic workload classes that are representative for applications of a certain domain.

6. RESEARCH DIRECTIONS

The approach of exhaustive, tailored performance testing has immediately shown its benefits when we applied it to the development of the persistence service. The proof of concept implementations of the *Measurelet* idea and the Benchmark Framework have demonstrated that it is possible to create and run a multitude of benchmark applications with limited effort. Based on the experience we gained through applying the presented approach, we discuss different research directions that we see in this area.

How to design the actual benchmark applications?
While the approach presented in this paper simplifies the construction and execution of benchmark applications, it is out of the scope of this work to determine how the design of the benchmarks should look like. In order to answer questions like „What are the usage profiles that I should test in my benchmark applications?“, „What are performance-critical data model designs?“, or „How to deal with caching effects that can bias the results?“, it requires sophisticated engineering methods. An example for such a method could be the automated derivation of benchmark applications based on real user monitoring data.

How to deal with the large parameter space?
The approach allows developers to create and run large sets of benchmark applications based on parametrisable *Measurelet* construction. However, the potential degrees of freedom for a developer of a benchmark application are huge. There are, for example, a vast amount of different potential data models. The same is true for the application logic and the usage profiles. Moreover, the number of test environments as well as the time available to execute the tests are limited. Thus, smart approaches are required that provide maximum information gain with a limited number of benchmark applications.

How to derive prediction models to provide performance feedback to application developers?
In Section 5 we introduced a scenario that aims at providing feedback to application developers with respect to the

performance of the persistence service for their specific application. However, deriving a prediction model that provides accurate predictions for different real-world applications is a challenging task. Besides the two questions stated above, one has to deal with finding an appropriate abstraction level that on the one hand limits the complexity of applications and platform service, but on the other hand includes enough detail to enable accurate predictions. A systematic search for proper heuristics in combination with sophisticated statistical analysis and machine learning techniques could be a potential direction towards answering this question.

How to apply the approach to other components? The idea of having a Measurelet API and a parametrisable framework that creates and runs benchmark applications is independent of JPA and the persistence service. Except for the JPA Benchmark Model Model introduced in Section 4.1 all the components presented in this paper are reusable for other performance measurements. However, so far we do not have any information on the effort for applying the approach to a different component (e.g., a messaging or identity management service). Moreover, other components can introduce additional challenges with respect to automated performance testing.

7. RELATED WORK

The approach presented in this paper combines automated model-based benchmarking and benchmark application generation with systematic measurements. In the following, we discuss existing work that addresses these aspects.

Benchmarking is a common and widely spread task in industry. Standard benchmarks, such as those offered by the Standard Performance Evaluation Corporation (SPEC) [15], the Transaction Processing Council (TPC) [18] (e.g. TPC-H and SPECjEnterprise2010), or software vendors like SAP [13], provide standardized performance measurement for comparing and ranking of platform configurations or platforms of different vendors. The JPA Performance Benchmark [9] is the largest benchmark that deals with JPA-based applications and allows the comparison of different JPA providers (e.g. EclipseLink, Hibernate) and Database Management Systems (e.g. Derby, MySQL). However, standard benchmarks do not satisfy individual information needs and are often difficult to setup. Therefore, specifically tailored and easily employable benchmark applications are needed to satisfy a certain information need.

Wright et al. [22] present Auto-Pilot a tool for the automation of performance measurements. Benchmark scripts execute performance measurements described using a procedural language. Auto-Pilot provides the ability to run analysis on performance measurements but does not use descriptive models for performance measurements in general. Kalibera [8] et al. present an approach for automatic performance measurements in a distributed and heterogeneous environment. They use an execution framework to take performance measurements and a benchmarking framework to control the execution. However, the execution framework requires the implementation of a certain run time environment and communication protocol. Moreover, Kalibera et al. do not use models to describe performance measurements and benchmark applications.

Apfelbaum and Doyle [2] as well as Malik et al. [11] present approaches for generating functional tests from models that describe the system under test. However, our JPA Benchmark Models describe performance measurements and not functional tests. In [24], Zhu et al. present a model-based approach for automated usage of performance measurements. The benchmark software is described in UML based models that are enhanced with annotations and elements describing the usage of the benchmark in order to conduct performance measurements. The model of the benchmark software can be derived automatically from UML models of the system under test with a model to model transformation. The model-based approach using UML for model description focuses on the integration of performance measurements in model-based software development. In contrast, our JPA Benchmark Model is domain specific and targets tailored measurements by describing test-relevant parts of the benchmark application.

In contrast to a model-based approach, Denaro [4] describes an approach for benchmark application construction based on existing software artefacts. Unimplemented parts are replaced by stubs which provide only minimal requirements (e.g. just an interface). The assumption is that the results delivered by the benchmark applications approximate the final performance characteristics of the software system. However, benchmark applications and required stubs have to be developed like common software system artefacts. Our benchmark framework complements the approach by automatically generating benchmark applications. The benefit of our model-based approach is to replace stubs with the concrete implementation of the artefact as development evolves.

8. CONCLUSIONS

In this paper, we introduced an approach that allows for exhaustive, tailored performance testing. The two main components of the approach are the *Measurelet* concept as well as the model-based construction and execution of benchmark applications. Using the presented approach, developers can create and run a large set of tailored benchmark applications with minimal effort. We have illustrated how we use the capabilities of the approach to ensure the quality of a JPA-based persistence service via detailed performance regression testing. Besides this platform service provider view, we also reported on how we plan to use the approach to provide valuable performance feedback to the consumer of the platform service (i.e., the application developers) in order to support them in developing high-quality applications. In our future work, we focus on implementing the performance feedback for the platform service consumer. Therefore, we are investigating sophisticated methods to derive performance prediction functions based on the systematic execution of parametrised benchmark applications. Moreover, we plan to adopt the exhaustive performance regression testing approach to other platform services.

Acknowledgment

This work is supported by the German Research Foundation (DFG), grant RE 1674/6-1 (Transfer project KIT-SAP).

9. REFERENCES

[1] Apache Software Foundation. The Apache Velocity Project. http://velocity.apache.org/, 2012.

[2] L. Apfelbaum and J. Doyle. Model based testing. *Software Quality Week Conference*, pages 1–14, 1997.

[3] L. DeMichiel and J. P. E. Group. JSR 317: Java Persistence API, Version 2.0. http://jcp.org/en/jsr/detail?id=317, 2009.

[4] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*, pages 94 – 103. ACM, 2004.

[5] Forschungszentrum Informatik (FZI). SQools: Software Analysis Tools. http://www.sqools.org/, 2012.

[6] Hibernate. Relational Persistence for Java and .NET. http://www.hibernate.org, 2012.

[7] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley, 1991.

[8] T. Kalibera. Regression benchmarking environment. *Proceedings of WDS*, 2004.

[9] I. Kirsh. JPA Performance Benchmark (JPAB). http://www.jpab.org/, 2012.

[10] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation Journal*, 2009.

[11] Q. A. Malik, J. Lilius, and L. Laibinis. Model-based testing using scenarios and event-b refinements. *Methods, Models and Tools for Fault Tolerance*, pages 177–195, 2009.

[12] R. Mordani. JSR 315: Java Servlet Specification, Version 3.0 Rev a. http://jcp.org/en/jsr/detail?id=315, 2010.

[13] SAP. Standard Application Benchmarks. http://www.sap.com/campaigns/benchmark, November 2011.

[14] SAP AG. Sap netweaver cloud. http://www.sap.com/solutions/technology/cloud/netweaver/index.epx, 2012.

[15] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org/, 2012.

[16] S. Stefanov. Book of speed: The business, psychology and technology of high-performance web apps. http://www.bookofspeed.com, 2012.

[17] The OSGi Alliance. OSGi service platform core specification. http://www.osgi.org/Specifications, 2012.

[18] TPC. Transaction Processing Performance Council. http://www.tpc.org/, 2012.

[19] D. Westermann. A generic methodology to derive domain-specific performance feedback for developers. In *Proc. of the 34th International Conference on Software Engineering (ICSE 2012), Doctoral Symposium*, 2012.

[20] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The Performance Cockpit Approach: A Framework for Systematic Performance Evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, pages 31–38. IEEE Computer Society, 2010.

[21] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM.

[22] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–19, 2005.

[23] G. York. EclipseLink JPA: An Advanced ORM Persistence Framework. Dzone Refcardz, http://refcardz.dzone.com/refcardz/eclipselink-jpa, 2011.

[24] L. Zhu, N. B. Bui, Y. Liu, and I. Gorton. MDABench: Customized benchmark generation using MDA. *Journal of Systems and Software*, 80(2):265–282, Feb. 2007.