Ginpex: Deriving Performance-relevant Infrastructure Properties Through Goal-oriented Experiments

Michael Hauck
FZI Research Center for Information Technology
Karlsruhe, Germany
hauck@fzi.de

Michael Kuperberg
Karlsruhe Institute of Technology
Karlsruhe, Germany
kuperberg@kit.edu

Nikolaus Huber
Karlsruhe Institute of Technology
Karlsruhe, Germany
nikolaus.huber@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
reussner@kit.edu

ABSTRACT

In software performance engineering, the infrastructure on which an application is running plays a crucial role when predicting the performance of the application. Thus, to yield accurate prediction results, performance-relevant properties and behaviour of the infrastructure have to be integrated into performance models. However, capturing these properties is a cumbersome and error-prone task, as it requires carefully engineered measurements and experiments. Existing approaches for creating infrastructure performance models require manual coding of these experiments, or ignore the detailed properties in the models. The contribution of this paper is the GINPEX approach, which introduces goal-oriented and model-based specification and generation of executable performance experiments for detecting and quantifying performance-relevant infrastructure properties. GINPEX provides a metamodel for experiment specification and comes with pre-defined experiment templates that provide automated experiment execution on the target platform and also automate the evaluation of the experiment results. We evaluate GINPEX using two case studies, where experiments are executed to detect the operating system scheduler timeslice length, and to quantify the CPU virtualization overhead for an application executed in a virtualized environment.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques,Performance attributes*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

QoSA+ISARCS'11, June 20–24, 2011, Boulder, Colorado, USA.

General Terms

Measurement, Performance

Keywords

Performance prediction, Experiments, Measurements, Infrastructure, Metamodel

1. INTRODUCTION

The performance of a software system depends on the performance of the underlying execution platform and on the workload. The increasing complexity of execution platforms (e.g. virtualized servers, load balancing or enhanced middleware) makes it more difficult to understand performance of software. At the same time, the existing execution platform performance models are too simple and no longer suitable for performance prediction [17]. Thus, when analyzing or predicting the performance of a software system, the execution platform has to be taken into account in more detail.

In the last decades, Software Performance Engineering (SPE [18]) supported software architects in analyzing the performance of software systems at different stages of the software lifecycle. Model-based SPE approaches, such as the Palladio Component Model [4] allow to generate performance models (e.g. queuing networks or Petri nets) of software systems from existing architectural models (components, behaviour specifications, etc.). Using such transformational approaches leads to reduced efforts, and also shields architects and developers from fine-grained mathematical formalisms of conventional performance models.

Although there exists a large body of research on abstracting performance models [2, 12], the task of *instantiating* them and the quantification of concrete model attributes values remains a manual, error-prone task. For modelling the applications, some automation has been achieved using machine learning and application instrumentation [13]. But when modelling the execution platform, the quantification of important attributes (such as operating system scheduling properties or different hardware resource characteristics) remains an open problem, since such attributes are mostly not specified and depend on the execution platform.

To ease the burden of integrating performance-relevant properties of the infrastructure environment into performance analysis, we propose an approach that detects such properties automatically through goal-oriented experiments, i.e. experiments that aim at inferring infrastructure properties with pre-defined experiment analysis logic. These experiments are executed on each relevant target platform. The experiments issue certain load patterns on the platform, observe the effect of patterns on the system's performance (response time, throughput, and resource utilisation), and infer properties based on the measured results.

In this paper, we present the Ginpex approach (Goal-oriented INfrastructure Performance EXperiments), which allows for specifying, executing and evaluating such goal-oriented experiments. Ginpex can be used by performance analysts to automatically derive performance-relevant infrastructure properties for performance predictions. For each property, an experiment can be specified in the tool together with analysis logic for evaluating the experiment results. Depending on the part of the infrastructure that is in the focus (e.g. operating system properties, hard disk or network resources), Ginpex has been designed to provide a library of experiments that can be selected for execution. Ginpex already contains pre-defined experiments for various infrastructure properties, and is kept extendable in order to add new experiments.

Ginpex can be used in model-based performance prediction approaches during design time, but also at later stages of the software lifecycle. For example, performance predictions can be conducted when parts of the infrastructure change (e.g. a major upgrade of the operating system kernel, migration of components to virtualized environments, or replacement of hard disk devices). In this case, little effort is necessary to re-run the experiments in order to detect the updated infrastructure properties that are needed for performance prediction.

In order to systematically set up new experiments, we developed a metamodel which is used for specifying the experiments in detail. Metamodelling the experiments has several advantages. First, a model-based approach provides an easy and elegant way to specify experiment configurations. Additional experiments and their specifications can easily be added by adding metamodel instances. Furthermore, existing modelling frameworks come with extensive model support. Based on the modelling framework, we derived modelling editors that can be used to display experiments in a convenient way, and allow for easily creating new experiments. The Ginpex framework also provides programmatic access to the model, which can be used for automated generation of experiment descriptions, rather than GUI-based DSL editing. By using a metamodel for the experiments, we separated the specification of an experiment from its execution: the (often platform-specific) way of how an experiment is executed and interpreted is not encoded in the model. In addition, the implementation and the metamodel of Ginpex itself can be extended in a systematic way, since we used established model-driven technologies and transformation languages.

The specified experiment model is transformed into executable Java classes which can be run on any suitable platform. The outcomes of the experiment runs are analysed by Ginpex to derive the desired attribute value from the experiment results. The goals can apply to various attributes and resources, e.g. operating system scheduler properties, virtualization platform properties, hard disks or network resources.

The contributions of this paper are (i) automated derivation of performance-relevant execution platform attribute values through the new Ginpex approach for goal-oriented experiments, (ii) a metamodel for the systematic specification of consistent Ginpex experiments, and (iii) reusable experiment series for two different infrastructure properties (detecting the operating system scheduler timeslice length, as well as quantifying CPU virtualization overheads).

We evaluate the contributions of the paper by defining experiments for detecting the timeslice length of operating system schedulers and for quantifying the CPU virtualization overhead of the Xen hypervisor. The validity of Ginpex results for the timeslice length is shown by comparing them to the vendor-specified values (Ginpex itself had no access to this information). The utility of Ginpex results for the virtualization overhead is shown by predicting the performance of an application based on the SPECjms2007 Benchmark: the prediction quality increases when Ginpex-provided overheads are included into performance models.

The remainder of this paper is structured as follows: Sec. 2 outlines the Ginpex approach, while Sec. 3 describes the core aspects of its implementation. Sec. 4 presents the evaluation using two case studies. Sec. 5 contrasts our approach with related work, and Sec. 6 concludes.

2. APPROACH

In this section, we give an introduction to the overall approach covered by the Ginpex tool. The approach aims at automatically deriving performance-relevant infrastructure properties based on goal-oriented measurements. It can be embedded into Software Performance Engineering (SPE) approaches to enhance model-based performance predictions.

An initial overview on the approach is given in Figure 1 and consists of the following steps: In the first step, a program has to be deployed on the machine (or machines) on which measurements are to be taken, which is called *Load Driver* in the following. The Load Driver is used for issuing load on the machine based on the experiment specification and taking the measurements. Once the Load Driver is deployed, experiments can be selected for execution. During execution of an experiment, different patterns of CPU and I/O load are issued by the Load Driver(s), and certain measurements (e.g. response times or CPU utilization) are taken for specific parts of the issued load. To derive performance-relevant properties, the load patterns in the experiments have to be designed in a way that the measured results allow to infer infrastructure properties through statistical analyses. In a third step, the measurement results serve as input for an analysis to derive the performance-relevant properties.

Finally, the detected properties are integrated into the performance prediction model. This can be done for example by using a configuration model. In this case, the detected properties would be passed to the prediction tool as a configuration instance. Once the performance prediction tool is configured based on the detected performance properties, the software architect can conduct a performance prediction that takes the experimentally derived infrastructure performance properties into account. The Ginpex tool explicitly covers the steps denoted by the grey boxes in Figure 1.

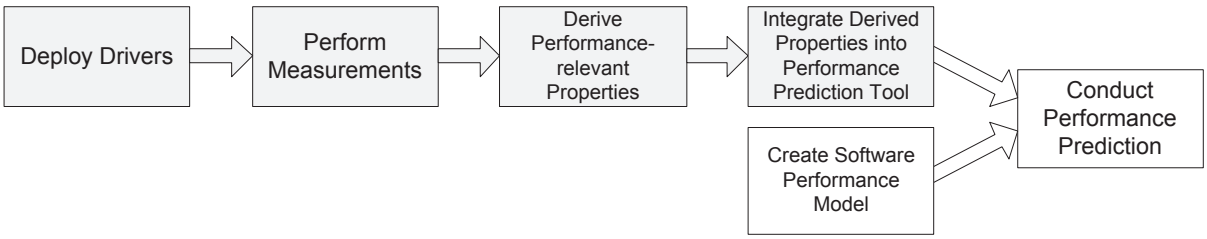


Figure 1: Overall Workflow of the Approach

Figure 2 shows the approach within the tool framework. The main part of GINPEX runs on the *controller machine*, which is not part of the target platform on which measurements are taken. Here, the user can either manually specify experiments and evaluate the results, or select pre-defined experiments which are then executed and evaluated automatically.

We call the actual specification of an experiment a *task set*. A task set specifies the control flow of the experiment, the load that should be generated during the experiment, and the sensors that indicate the parts where measurements have to be taken. An experiment task set might be executed on different machines, for example in order to examine parameters of a network connection between two machines. To indicate which part of the experiment is being executed on which machine, several *machine task sets* can be specified in a task set.

The machine task sets of an experiment are being transformed into executable code and transferred to the machine(s) for execution. The GINPEX tool ships with a Load Driver that has to be run on the target machine(s). The Load Driver executes the experiment code, manages the measurements to be taken and transfers the measurement results back to the controller machine. Upon experiment completion, the controller machine stores the experiment results. In the manual case, the user can inspect the experiment results for further reasoning. In the automated case, the pre-defined experiments contain logic for analysing the experiment results. This logic derives the infrastructure properties and exports them into a configuration model instance that can be used in the performance prediction tool.

We are using GINPEX to detect infrastructure performance properties that are used for performance prediction with the Palladio Component Model (PCM) [4]. By using a configuration model to export the detected infrastructure properties, automated configuration of the PCM performance analysis tooling is possible. To use a different performance prediction tool, only the output format of the detected properties would have to be adapted.

The GINPEX tool can be downloaded from the web [6].

3. MODEL-BASED EXPERIMENT DEFINITION AND EXECUTION

In the following, we explain the approach in more detail. We give an overview on the metamodel which we developed for experiment and task specification in GINPEX and explain how experiments are executed.

3.1 Experiment Library and Experiment Domains

GINPEX provides pre-defined experiments together with evaluation logic for the results of these experiments.

Every experiment aims at deriving one characteristic of an infrastructure property. This property is called the *detected parameter* of the experiment. In some cases, an experiment might need additional parameters as input configuration. If an input parameter is known to the user, he can specify the parameter prior to the experiment execution. Otherwise, if a pre-defined experiment is available that detects this parameter, this experiment is executed before the selected experiment is being executed. Note that GINPEX requires the

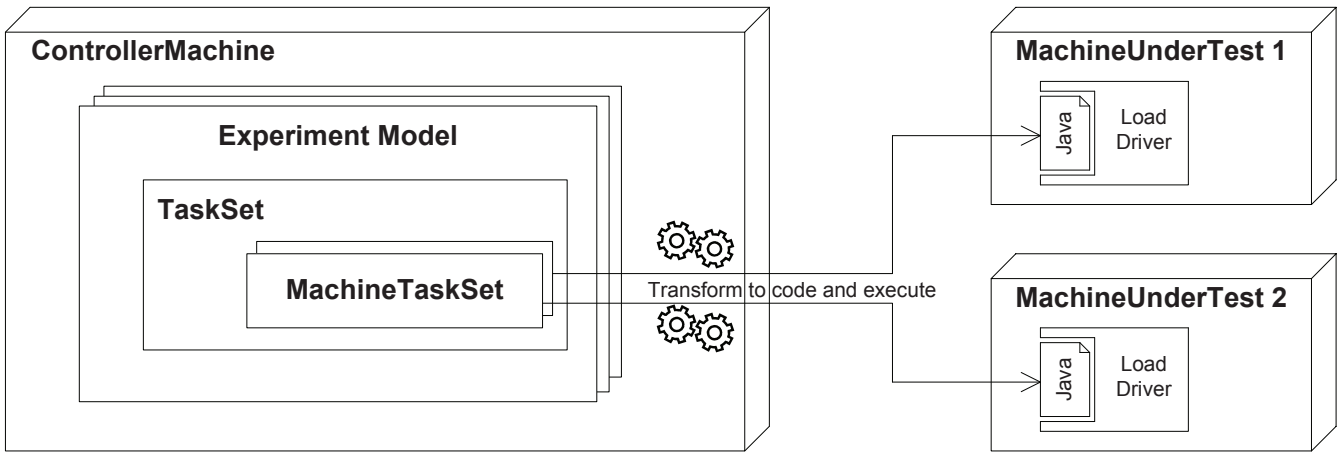


Figure 2: Examplary Experiment Machine Setup

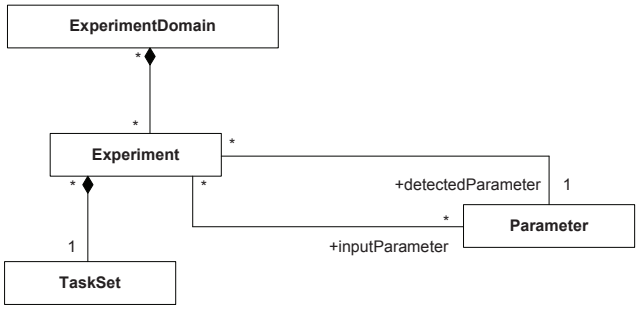


Figure 3: Ginpex Experiments Metamodel

specification of a partial order on parameters, i.e. there must not be a cycle in depending parameters.

As Ginpex can be applied to various parts of the infrastructure, pre-defined experiments are grouped in *experiment domains*. An experiment domain specifies a certain part of the infrastructure whose properties are to be detected. For example, certain performance predictions may be conducted which only focus on the hard disk performance of an application and are only interested in reflecting hard disk properties (such as disk cache performance impact). In this case, the prediction abstracts from properties of other parts of the infrastructure, and thus, only experiments for the experiment domain of hard disks are of interest. Other experiment domains include network properties, OS scheduling properties, and virtualization properties. For automated experiments, the user can either select a complete experiment domain, or a subset of experiments for an experiment domain for execution.

The structuring of experiments within the Ginpex metamodel is shown in Figure 3.

3.2 Experiment Specification

Pre-defined experiments as well as manually defined experiments share the same metamodel structure specifying the experiment task set. An exemplary task set and its GUI visualization can be found in Section 4.1 (see Figure 6). In the following, this part of the metamodel is described in more detail.

Figure 4 shows the Ginpex task set metamodel, Figure 6 shows the GUI support for specifying a task set and its contents. The central element of the experiment specification is the `TaskSet`. A `TaskSet` is made up of three different parts: (i) a specification of target machines on which the experiment is to be executed, (ii) the tasks that have to be executed, and (iii) sensors that indicate which measurements have to be taken for which tasks.

All executable tasks inherit from `AbstractTask` and are presented in more detail below. Sensors reference the task for which measurements have to be taken. Currently, two different types of sensors are supported. A `ResponseTimeSensor` measures the time a task needs for execution, a `CpuUtilizationSensor` measures the overall CPU utilization of the machine while the task is being executed. Both types of sensors can be specified for any kind of task.

An overview on all executable tasks is given in Figure 5. For more detailed information, please refer to the Ginpex metamodel documentation at [6]. Executable tasks fall into two groups. The first group consists of tasks that specify the control flow of the experiment. Control flow tasks contain nested tasks that are to be executed in a certain way. A `SequenceTask` executes all included task one after another, while a `ParallelTask` executes all included tasks concurrently. Additionally, for a `ParallelTask` the attribute `stopAfterFirstTaskCompleted` indicates whether the `ParallelTask` should abort nested tasks once the first nested task has completed. `ParallelProcessTask` inherits from `ParallelTask`. In contrast to `ParallelTask`, which uses threads for parallel execution of tasks, a `ParallelProcessTask` forks child processes for parallel execution. A `LoopTask` executed a nested task multiple times. The stop condition of a `LoopTask` can be modelled with different elements that inherit from the abstract `StopCondition` element (not shown in the figure). Such elements allow for example specifying loops that run a specific number of iterations, or endless loops. Finally, a `MachineTaskSet` denotes another type of task which contains tasks that are to be executed on a certain target machine. By modelling task sets containing multiple `MachineTaskSet`s which reference different `MachineReference`s, the experiment execution can be distributed to different target machines. Note that only tasks that are modelled within a `MachineTaskSet` are actually executed on a target machine. All other tasks are executed on the controller machine.

Apart from control flow tasks, machine tasks denote the actions that are to be executed during the experiment. Such tasks have to be nested inside a `MachineTaskSet`, and thus, can only be executed on target machines. Currently, such tasks include the generation of certain types of load on the target platform. For example, a `CpuLoadTask` can be specified to put different types of CPU load on the platform. A `NetworkLoadTask` creates network load that is sent to a different target machine (referenced through the `receivingMachine` association).

Further specification details, such as model attributes, are omitted from the figures for the sake of brevity. However, we explain the `CpuLoadTask` model element and its attributes in more detail in the following.

The `CpuLoadTask` allows for generating CPU load executed in a single thread. It contains two different attributes for specification: The amount of demand specified by the `duration` attribute, and the type of demand specified by the `demand` attribute. The type of demand can be chosen out of `MandelbrotDemand`, `FibonacciDemand`, `SortArrayDemand`, and `WaitDemand`. `MandelbrotDemand` and `FibonacciDemand` generate load that puts more stress on the CPU than on the memory. In contrast, `SortArrayDemand` issues load performing sorting logic that strongly affects memory. `WaitDemand` does not directly issue CPU load, but waits for the specified duration, and thus can be seen as an "empty demand" type.

The specified duration denotes the time in milliseconds that the demand would take to execute on the target machine on a single core without contention. This is achieved by calibrating resource demand measurements on the platform prior to experiments execution. This calibration determines input parameters for the load generation algorithms to match the specified duration times. The detailed approach is explained in [3].

3.3 Experiment Execution

To execute an experiment, we decided to generate executable code based on the `MachineTaskSet`s instead of in-

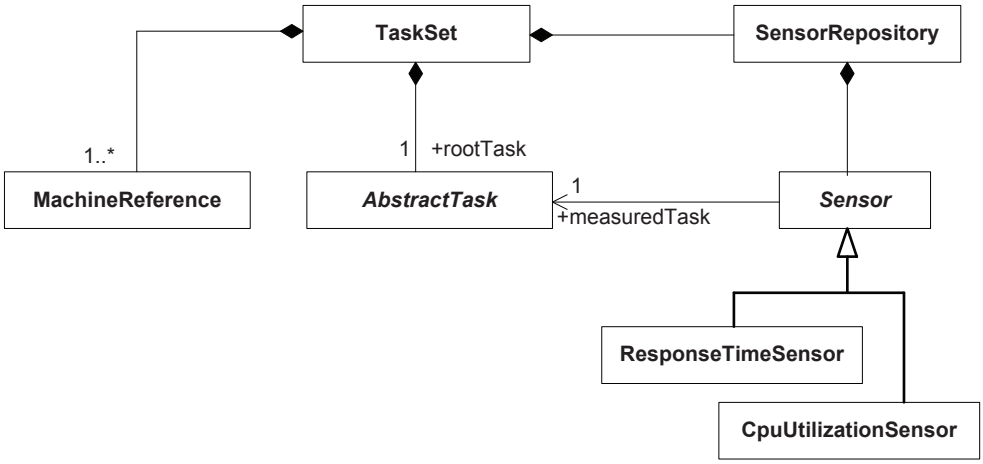


Figure 4: Ginpex Task Set Metamodel

terpreting the models in order to keep the overhead of performing measurements low. For each `MachineTaskSet`, Java code is being generated that conforms to the specified tasks inside the `MachineTaskSet`. The generated code includes for each task the task logic to be executed, as well as sensor logic (i.e. response time and/or CPU utilization measurements), if sensors had been specified for the tasks.

The generated code is being transferred to the Load Driver running on the machine references by the `MachineTaskSet`. The Load Driver then compiles the code and runs the initial preparation part of the experiment. This part includes the result objects initialization, as well as different checks (for example if needed network connections are working etc.).

For `CpuLoadTask`s, the preparation phase also includes the calibration step explained above. As calibration might take some time, the calibration results are stored in calibration files which are then used in later runs. Ginpex also supports executing `CpuLoadTask`s on a machine while using calibration files from a different machine. Through this, the exact same amount of load can be put on different machines. For example, if specifying `CpuLoadTask`s with 50 ms on two different machines, a 1 GHz CPU machine and a 2 GHz CPU machine, the tasks should take around 50 ms on both machines if the calibration files are generated individually on each machine. If the calibration file from the 1 GHz CPU machine is reused for executing the task on the 2 GHz CPU machine, the task should run in around 25 ms on this machine. Sharing calibration files across machines is useful for detecting machine speedups compared to reference machines. We used this approach in the virtualization overhead case study in Section 4.2.

After the code for all `MachineTaskSet`s has been generated and prepared, the experiment is being executed. Depending on the specified experiment control flow, the controller machine asks the Load Drivers to execute the corresponding `MachineTaskSet`s. Once the overall experiment control flow is completed, each Load Driver reports all measurement results for the executed `MachineTaskSet`s to the controller machine.

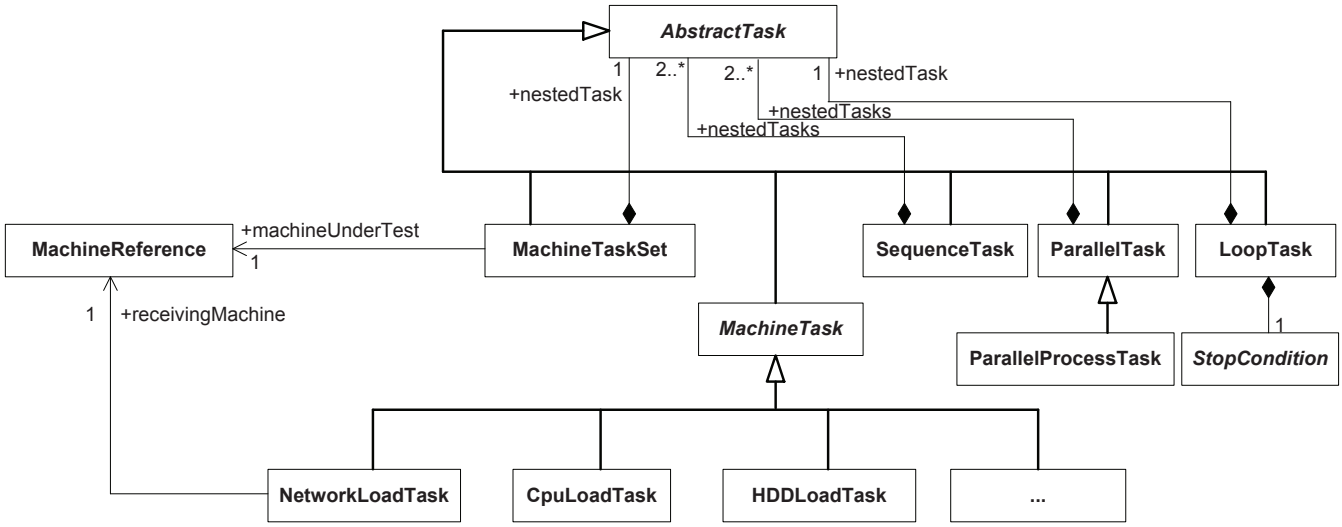


Figure 5: Ginpex Tasks Metamodel

3.4 Results Analysis

After experiment completion, the measured results are available to the controller machine and can be visualized to the user. If the evaluation logic is available (as it is the case for pre-defined experiments), the evaluation phase directly follows the experiment execution.

In this step, the experiment results are analyzed in order to derive performance properties for later performance predictions. GINPEX provides access to statistical libraries, for example the R statistical computing engine, which facilitates enhancing GINPEX with additional experiment analysis logic. More details on GINPEX extendability are given in Section 3.5.

The output format of the detected parameters depends on the scenario in which GINPEX is used. In our work, we use GINPEX in the scope of model-based performance prediction with the Palladio Component Model (PCM) [4]. To integrate infrastructure properties into performance prediction, GINPEX stores the detected properties in a configuration model that can be used by the PCM for performance prediction. However, the GINPEX is not linked to a specific performance prediction approach, but can also be enhanced to export output parameters in a different format.

3.5 Ginpex Extendability

The GINPEX tool has been designed to be extendable w.r.t. (i) experiment domains, (ii) experiments, and (iii) experiment tasks. GINPEX is integrated into the Eclipse IDE platform and makes use of the Eclipse OSGi architecture [20]. For experiment domains and experiments, Eclipse extension points have been defined, i.e. new experiment domains and experiments can be added by specifying the corresponding Eclipse extensions. GINPEX provides an API that indicates the methods that have to be implemented by new experiments. Such methods include the specification of the input and output parameters, the generation of the experiment model, and the analysis logic of the experiment results.

Adding a new experiment specification and the corresponding analysis logic certainly requires detailed domain knowledge. The experiment has to be defined in a way to detect infrastructure properties without being tailored towards a certain platform, as the experiment should be applicable to a variety of platforms. For example, specifying an experiment that detects the throughput of write requests to a storage device should be applicable to platforms equipped with hard disks as well as solid state disks. As GINPEX also can be used to perform experiments manually, it supports the developer in performing explorative measurements which precede the definition of new experiments and experiment analyses.

Enhancing experiment tasks can be done by using established model-driven technologies included in the Eclipse platform. The experiment metamodel is specified in Ecore. Using the Eclipse Modeling Framework, developers can add new tasks to the metamodel. In order to support new tasks in experiment execution on the Load Driver, code generation templates have to be enhanced with Java code which is to be generated for the added task. For adding a new task, four templates are available for specifying Java code to be generated:

- The `VariablesDeclaration` template contains all variable definition that the task accesses.
- The `TaskPreparation` template contains preparation logic. This logic is called prior to the task execution in the preparation phase.
- The `TaskExecution` template contains the actual execution logic. This logic is called in the execution phase.
- The `TaskCleanup` template contains cleanup logic which is called after the experiment has been executed.

The templates are specified using the Xpand language of the Eclipse Model-To-Text (M2T) framework [21]. In addition, we plan to extend GINPEX so that existing Java libraries can be easily incorporated into experiments, without the need to enhance or change the meta-model or the code generation templates.

4. CASE STUDY

In the following we present the results of two case studies in which we applied GINPEX in order to detect performance-relevant properties. We chose two different domains for the case studies to demonstrate that GINPEX is not restricted to a certain domain of the infrastructure a software is running on. The first case study focusses on detecting the timeslice length property of the operating system scheduler and gives an example for an experiment model instance. In the second case study, GINPEX was used to predict the performance impact of CPU virtualization overheads when migrating a software system to a virtualized environment.

4.1 OS Scheduler Timeslice Length

Detecting the time slice length of operating system schedulers is part of the experiment domain "OS scheduling" that is included in GINPEX. The timeslice of an OS scheduler is the amount of time that the scheduler allows a task to work on the CPU without interruptions when there is contention on the CPU. In the "OS scheduling" domain, additional experiments have been defined that aim at detecting load-balancing properties of the OS scheduler. These experiments are available as pre-defined experiments in GINPEX. Detailed experiment descriptions, experiment results and its impact on performance prediction results can be found in [8].

Reflecting scheduler timeslices in performance prediction is crucial, as especially for short requests, abstracting from the timeslice can lead to imprecise predictions. The results gained in [8] actually suffer from neglecting the timeslice length in a performance prediction simulation.

The experiment described in the following is pre-defined in GINPEX and can thus be used out-of-the-box in order to detect the timeslice length on an arbitrary platform. The experiment for detecting the timeslice length depends on one input parameter, which denotes the number of CPU cores that is available on the target machine. While GINPEX also provides an experiment to detect the number of cores, this property often is known to the user and can be specified prior to the experiment run.

The experiment for detecting the timeslice length is defined as follows: In parallel running processes, CPU load is issued on the machine. The number of parallel processes is twice as high as the number of available cores, so that every parallel task issuing CPU load is assumed to share a core with another task. One of the tasks repeatedly issues small amounts of CPU load, i.e. 20 ms CPU load. Between the demands, response time measurements are taken. For

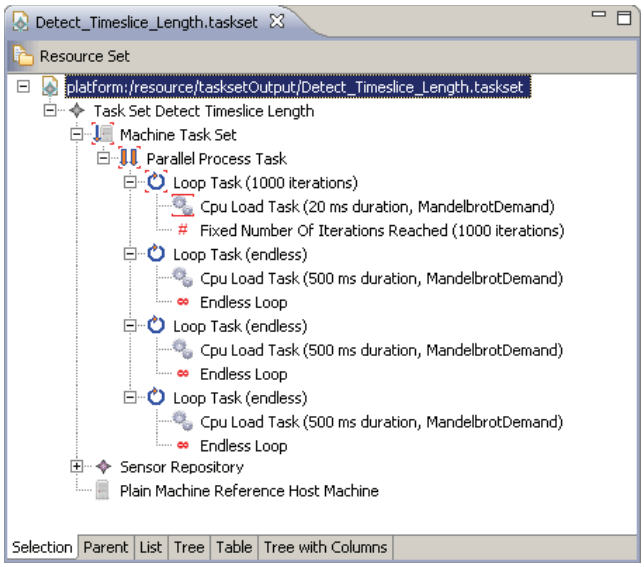


Figure 6: Task Set for Detecting the OS Timeslice Length on a Dual-core Machine

this experiment, we assume that the actual timeslice length is larger than 20 ms in order to yield measurements that can be used for analysis. However, given the fact that all common operating systems use average timeslice lengths between 30 ms and 500 ms, we believe this is a valid assumption.

The remaining tasks also continuously issue CPU demands in order to put load on the CPUs. On these tasks, no measurements are taken, thus, only measurement results for the first task are regarded. The experiment assumes that the measured task results fall into two clusters. One portion of results is expected to be approximately around 20 ms, i.e. the scheduler put the executed demand into one timeslice. The remaining results are expected to be a lot higher than 20 ms. In this case, the scheduler has interrupted the executed demand and put the parallel running task on the CPU. As all tasks are running with the same priority, we assume that the task has been interrupted for exactly one timeslice. The measured result in this case includes the execution of the 20 ms demand, in addition to the interruption time of one timeslice. The difference between the average measured time of the first cluster and the average measured time of the second cluster can be interpreted as the timeslice length. Ginpex uses a clustering algorithm [9] for this experiment, which is a derivation of the k-means clustering method.

We executed the experiment on a dual-core machine[1] on three different operating system environments (Windows XP, Ubuntu Linux kernel 2.6.22, Fedora 12 Linux kernel 2.6.31), whose scheduler implementation differs in the timeslice length. Figure 6 shows a screenshot of the experiment editor displaying the generated task set for the experiment. As the experiment is executed on a dual-core machine, in total 4 parallel nested tasks have been created. The bars around the first CPU Load Task indicate that response time sensors have been defined for this task.

Figure 7 shows the experiment results on the three systems. For each system, the results of the measured tasks are shown as cumulative distribution function (CDF). For each result, the two clusters are clearly visible. The difference in the average results in each cluster yields a calculated timeslice of 31 ms for Windows XP, 100 ms for Linux 2.6.22, and 50 ms for Linux 2.6.31.

According to the OS scheduling documentation [16, 1, 15], the actual average scheduler timeslice length is 31.5 ms for Windows XP, 100 ms for Linux 2.6.22, and 50 ms for Linux 2.6.31. Thus, the detected timeslice length matches the actual length accurately. We are aware that in certain operating systems, the timeslice length is being adjusted dynamically depending on the priority of processes. The priority management of OS processes has to be reflected in further experiments which are subject to future work.

4.2 Virtualization Overhead

In a second case study, we applied Ginpex to a supply chain management (SCM) system for supermarkets based on the SPECjms2007 Benchmark scenario described in [19]. The focus of this case study lies on detecting CPU overheads that occur when executing software in a virtualized environment. In the SCM system, we concentrate on three request types that occur in the system, static web page requests, online monitoring requests, and business intelligence report requests. We assume a user workload with an arrival rate of 500 requests per minute. Details on the workload mix are given in Table 1.

Table 1: Workload of the SCM Case Study

Request Type	Mean Service Time	Relative Frequency
Web Page	20 ms	85%
Monitoring	250 ms	10%
Reporting	2000 ms	5%

In the original setting, the case study is deployed on a single quad-core server[2] running Windows 7. Prediction models already exist that match the system with sufficient accuracy (see Table 2).

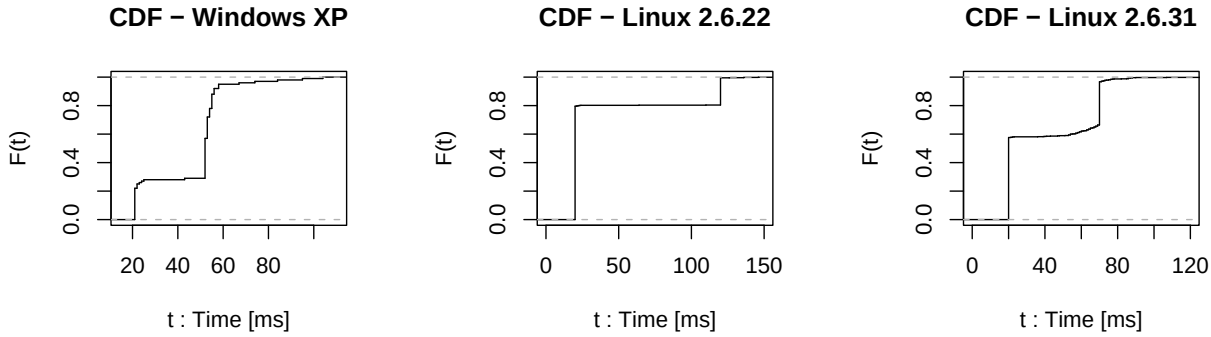
Table 2: Initial Prediction and Measurement Results for the SCM Case Study

	Average Measured	Average Predicted
Web Page Response Time	21 ms	20 ms
Monitoring Response Time	264 ms	251 ms
Reporting Response Time	2050 ms	2013 ms
Server Utilization	31.6%	29.7%

In a different setting, the SCM system is to be migrated to a distributed system, using virtualized servers. While the component that deals with the web page requests should remain on the native quad-core server, the Monitoring and Reporting components are to be migrated to separate virtualized servers. The new system consists of two native quad-core servers with the same hardware equipment as the server in the original setting. The first server runs Windows 7 and only hosts the Web Page component. The second server runs a virtualization hypervisor (XenServer 5.6) with two virtual machines. The first virtual machine runs Windows 7 and hosts the Monitoring component. The second virtual

[1]Intel Core 2 Duo, 2.66 GHz, 3 GB RAM

[2]Intel Core i7-860, 2.80 GHz, 8 GB RAM



(a) Timeslice Experiment Results on Windows XP.

(b) Timeslice Experiment Results on Linux 2.6.22

(c) Timeslice Experiment Results on Linux 2.6.31.

Figure 7: Timeslice Experiment Results on a Dual-core System with Windows XP, Linux 2.6.22, and Linux 2.6.31

machine runs Fedora 12 Linux (kernel 2.6.31) and hosts the Reporting component. Both virtual machines have been assigned two CPU cores and 2 GB RAM. No further virtualization adjustments, such as virtual CPU priorities, core pinning, or dynamic RAM adjustments, have been set.

The experiment for detecting virtualization overhead is structured as follows. The experiment consists of two parts, which are executed one after another (thus, the root task of the experiment is a `SequenceTask`). In the first part, a `MachineTaskSet` executes tasks on a non-virtualized machine which forms the base for the later virtualization overhead measurements. In the case study, this machine is the quad-core server on which all components are running in the original setting. Currently, the experiment first executes a `CpuLoadTask` multiple times without additional load on the machine. This task issues CPU demands with a duration of 500 ms and measures the duration time. In the second part, another `MachineTaskSet` executes tasks on the virtualized machine. It executes the same tasks as before. In order to put the same amount of load on the machine, this `MachineTaskSet` is modelled to use the calibration files from the non-virtualized machine for the `CpuLoadTask`s, as explained in Section 3.3.

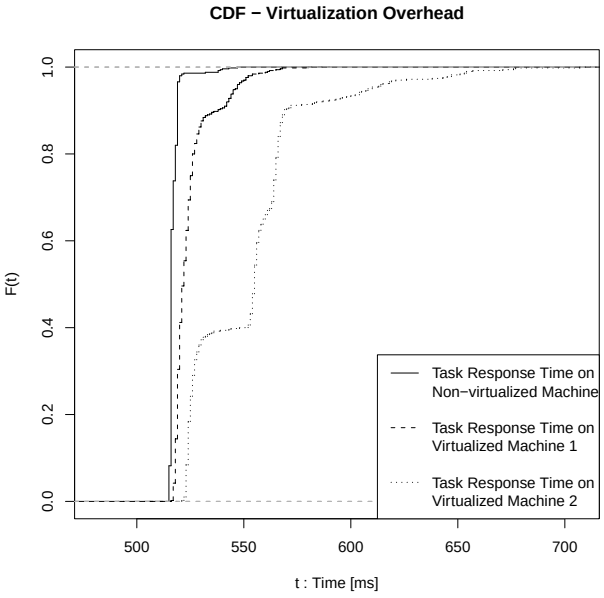


Figure 8: Results of the Virtualization Overhead Experiment

Figure 8 shows the response times for the three executed `CpuLoadTask`s. The response times of the task executed in a virtual machine are lower than the response times measured on the non-virtualized machine. From the results, a virtualization overhead of 1.3% can be calculated for virtual machine 1 and 6.7% for virtual machine 2. The different overhead results can be explained by the different guest operating systems that are used in the virtual machines.

Based on the results, the virtualization overheads have been integrated into the performance prediction models for the case study. The modelled infrastructure of the case study has been adapted in order to reflect the new target environment. The Monitoring and Reporting machines have been deployed on the virtual servers. In the performance analysis, the calculated overheads are taken into account by adding the overhead on every demand that occurs by a component deployed on the virtual machine.

Finally, we conducted a performance prediction with the adapted performance models and compared the predicted results with measurements taken from the case study components in the distributed setting.

Table 3 lists the measured and predicted average component response times and server CPU utilizations for the distributed case study scenario. In the last column, we depicted the prediction results for the prediction without taking into account the detected overheads. For all prediction, the respective prediction error is shown in brackets. The results show that prediction accuracy can be enhanced by taking into account virtualization overheads. Especially for virtual machines that incur a higher overhead, the prediction error is considerably lower.

Table 3: Prediction and Measurement Results for the SCM Case Study in the Distributed Setting. In Brackets: Deviation Percentage between Prediction and Measurement

	Average Measured	Average Predicted	Average Predicted w/o Overheads
Web Page Response Time (Non-virtualized)	20.02 ms	20 ms (0%)	20 ms (0%)
Monitoring Response Time (Virtual Machine 1)	272 ms	253 ms (7.0%)	250 ms (8.1%)
Reporting Response Time (Virtual Machine 2)	2616 ms	2541 ms (2.9%)	2314 ms (11.5%)
Non-virtualized Server Utilization	3.34%	3.5% (4.7%)	3.5% (4.7%)
Virtual Machine 1 Utilization	12.33%	10.1% (18.1%)	9.1% (26.2%)
Virtual Machine 2 Utilization	48.8%	48% (1.7%)	42.3% (13.3%)

Note that this case study only features rudimentary experiments on virtualization overhead in order to show the applicability of Ginpex and the variety of possible experiments. More experiments are necessary in order to detect a fine-grained model of the virtualization platform that cover different virtualization properties, such as the performance in scenarios with higher load, as well as the performance impact of I/O access within a virtual machine. Research on such experiments is left to future work.

5. RELATED WORK

In this section, we summarize related work deriving performance-relevant infrastructure properties through automated measurements and analysis.

Cherkasova and Gardner [5] use measurements to detect I/O overhead that occurs for the Xen virtualization platform. They propose a measuring and monitoring framework, but the approach is only applicable to the Xen hypervisor. Wood et al. [24] use microbenchmarks to estimate performance overheads that occur when migrating an application to a virtualized environment. They regard different kinds of virtualization overhead, but only focus on analyzing resource requirements, not on changes of response times. Iosup et al. [10] aim at measuring performance properties of cloud computing platforms. This approach focuses on cloud computing services for scientific computing. However, the measurement results are not used for integration into analysis tools, e.g. for performance prediction.

Happe et al. [7] developed a performance completion for message-oriented middleware to include low level platform details into performance prediction. The approach does not aim at enhancing the analysis tools, but focusses on extending a software architecture model by integrating a performance completion provided by a completion library.

Liu et al. [14] aim at predicting the performance of software running on a middleware platform. Performance-relevant properties of a Java EE platform are derived based on measurements, but underlying effects, such as hardware contention or operating system scheduling, are included in the benchmark results and not analyzed. Hence, the underlying platform cannot be decoupled from the middleware model. Additionally, the approach lacks tool support. In [25, 26], Zhu et al. extend the approach by a model-driven benchmark generation tool. The authors focus on benchmarks that are generated based on a software architecture description, whereas our approach employs model-driven load generation of infrastructure experiments agnostic to the software system under prediction.

Frameworks that do not focus on a specific part of the infrastructure include the ones presented by Kalibera et al. [11] and Tsouloupas and Dikaiakos [22]. These frameworks facilitate the automation of benchmark execution in distributed environments or grid environments, but do not cover automated evaluation of measurement results.

Another generic framework to conduct performance analyses is presented in [23]. This framework allows adding adapters to benchmark, monitor, and analyze the performance of a system. However, it is designed for automated measurement of whole (software) systems and does not focus on fine-grained analysis of infrastructure properties.

6. CONCLUSIONS

In this paper, we presented the Ginpex approach and tool for detecting performance-relevant infrastructure properties based on goal-oriented experiments. The experiments issue certain load patterns on the target platform and measure performance metrics for certain parts of the issued load. Afterwards, the measurement results are evaluated in order to derive properties that are not directly measurable.

Ginpex features a metamodel for modelling experiments and experiment task sets, which specify the actual execution logic of the experiment. Ginpex allows for storing predefined experiments together with experiment analysis logic that can be executed automatically on the target platform. We illustrated the applicability of Ginpex in two different case studies, focussing on the detection of operating system scheduler timeslice length and CPU overhead in virtualized environments. Ginpex can be downloaded from the web, where more documentation about the approach and the Ginpex metamodel is also available [6].

For future work, we plan to extend the pre-defined experiments and enhance Ginpex with further experiment scenarios. We want to add more pre-defined experiments to Ginpex so that Ginpex can also serve as an open platform for a large community to collect a variety of pre-defined experiments.

We are currently working on defining experiments for detecting hard disk properties and network connection properties in order to enhance software performance prediction with more accurate hard disk and network resource models.

Besides, we plan to further enhance experiments for virtualized environments to include further performance-relevant properties of virtualization platforms into performance prediction models, such as hypervisor scheduling properties or I/O access overhead. Based on the results gained in the conducted case studies, we believe that Ginpex provides a proper fundament for future research.

Acknowledgments

The work presented in this paper was partially developed in the context of EMERGENT: Grundlagen emergenter Software that is funded by the German Federal Ministry of Education and Research (BMBF) under grant 01IC10S01A.

7. REFERENCES

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. Technical Report, Silicon Graphics, Inc. (SGI), 2005.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[3] S. Becker, T. Dencker, and J. Happe. Model-Driven Generation of Performance Prototypes. In *SIPEW 2008: SPEC International Performance Evaluation Workshop*, volume 5119 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag Berlin Heidelberg, 2008.

[4] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[5] L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *USENIX 2005: Proceedings of the USENIX Annual Technical Conference*, 2005.

[6] Ginpex website. http://sdqweb.ipd.kit.edu/Ginpex.

[7] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner. Parametric Performance Completions for Model-Driven Performance Prediction. *Performance Evaluation*, 67(8):694–716, 2010.

[8] M. Hauck, J. Happe, and R. H. Reussner. Automatic Derivation of Performance Prediction Models for Load-balancing Properties Based on Goal-oriented Measurements. In *MASCOTS 2010: Proceedings of the 18th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society, 2010.

[9] L. J. Heyer, S. Kruglyak, and S. Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 1999.9:1106–1115, 1999.

[10] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2011.

[11] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban. Automated Benchmarking and Analysis Tool. In *VALUETOOLS 2006: Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*. ACM, 2006.

[12] H. Koziolek. Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation*, 67(8):634–658, 2010.

[13] K. Krogmann, M. Kuperberg, and R. Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 36:865–877, 2010.

[14] Y. Liu, A. Fekete, and I. Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11):928–941, 2005.

[15] I. Molnar. Linux: The Completely Fair Scheduler. http://kerneltrap.org/node/8059/, 2007.

[16] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals : Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 4. ed. edition, 2005.

[17] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of NSDI'06*, pages 239–252, Berkeley, CA, USA, 2006. USENIX Association.

[18] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, Reading, MA, USA, 1990.

[19] SPEC. SPECjms2007 Benchmark. http://www.spec.org/jms2007/.

[20] The Eclipse Foundation. Eclipse Equinox OSGi. http://www.eclipse.org/equinox/.

[21] The Eclipse Foundation. Eclipse Model To Text (M2T) Framework. http://www.eclipse.org/modeling/m2t/.

[22] G. Tsouloupas and M. D. Dikaiakos. Characterization of Computational Grid Resources Using Low-Level Benchmarks. In *E-SCIENCE 2006: Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*. IEEE Computer Society, 2006.

[23] 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)*. IEEE Computer Society, 2010.

[24] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Middleware 2008: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag, 2008.

[25] 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, 2007.

[26] L. Zhu, Y. Liu, N. B. Bui, and I. Gorton. Revel8or: Model Driven Capacity Planning Tool Suite. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 797–800, Washington, DC, USA, 2007. IEEE Computer Society.