Performance Modeling in Industry: A Case Study on Storage Virtualization

Nikolaus Huber
Karlsruhe Institute of Technology, IPD
76131 Karlsruhe, Germany
nikolaus.huber@kit.edu

Steffen Becker
FZI Forschungszentrum Informatik
76131 Karlsruhe, Germany
sbecker@fzi.de

Christoph Rathfelder
FZI Forschungszentrum Informatik
76131 Karlsruhe, Germany
rathfelder@fzi.de

Jochen Schweflinghaus
IBM Research and Development GmbH
71032 Boeblingen, Germany
schwefel@de.ibm.com

Ralf H. Reussner
Karlsruhe Institute of Technology, IPD
76131 Karlsruhe, Germany
reussner@kit.edu

ABSTRACT

In software engineering, performance and the integration of performance analysis methodologies gain increasing importance, especially for complex systems. Well-developed methods and tools can predict non-functional performance properties like response time or resource utilization in early design stages, thus promising time and cost savings. However, as performance modeling and performance prediction is still a young research area, the methods are not yet well-established and in wide-spread industrial use. This work is a case study of the applicability of the Palladio Component Model as a performance prediction method in an industrial environment. We model and analyze different design alternatives for storage virtualization on an IBM* system. The model calibration, validation and evaluation is based on data measured on a System z9* as a proof of concept. The results show that performance predictions can identify performance bottlenecks and evaluate design alternatives in early stages of system development. The experiences gained were that performance modeling helps to understand and analyze a system. Hence, this case study substantiates that performance modeling is applicable in industry and a valuable method for evaluating design decisions.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies, Modeling techniques; I.6.4 [**Model Validation and Analysis**]

General Terms

Performance, Measurement

*Trademarks of IBM in USA and/or other countries

ICSE '10, May 2-8 2010, Cape Town, South Africa

1. INTRODUCTION

Today, (software) systems consist of an increasing number of interacting components. The resulting complexity leads to an increasing difficulty of estimating and predicting non-functional properties. Moreover, non-functional requirements like the system performance are often considered at a late stage during development. However, early evaluation and prediction of the performance of a system can safe time and money for late redesigns [22].

Developing performance models requires time, knowledge and experience, thus increasing the initial cost of system development. Therefore, it is necessary to research effective tools and methods which ease and support the performance modeling and analysis process. However, none of the currently available approaches has gained widespread industrial use and many companies still rely on their software architects' experience instead of engineering methods [9, 12, 23]. Hence, it is important to demonstrate that methods and tools for software performance engineering evolve and improve software system development.

One particular approach for performance analysis is the Palladio Component Model (PCM) [4]. Its applicability and usability was investigated in theoretical and industrial context [1, 10, 12]. However, these investigations settled within the PCM's domain of business information systems which makes a generalization of the benefits of software performance engineering difficult.

This paper presents a case study on using the PCM to build a performance model for evaluating different storage hardware virtualization design alternatives for IBM systems. Furthermore, the intention of this case study is also to examine the applicability of the PCM in an industrial context *outside* its target domain of business information systems. Questions answered by this performance model are, which design alternative to choose, what the influences of additional resource provisioning are, which parts are potential performance bottlenecks and how can the system performance be improved.

The contributions of this paper are: i) a report on the calibration and validation of a performance model based on measurements on an IBM System z9, ii) an analysis and evaluation of different design alternatives for storage vir-

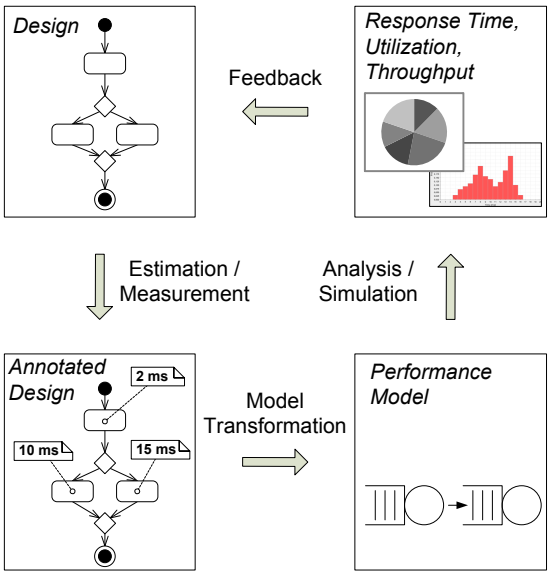


Figure 1: Model-driven performance engineering.

tualization and iii) a list of experiences we gained in this industrial case study. These experiences cover software performance engineering in general as well as the applicability of the PCM in the area of virtualized systems. The results demonstrate that the PCM is mature enough to support system development and predict the performance of a system.

Section 2 gives an overview of software performance engineering and the PCM, and explains the architecture of the modeled system. Section 3 describes the model and its calibration. In Section 4, the results are evaluated and discussed. Section 5 lists the experiences we gained. Section 6 discusses related work. Section 7 concludes this paper and presents an outlook for future work.

2. FOUNDATIONS

The following gives an overview of software performance engineering in general and the PCM approach in particular. Furthermore, the modeled system is introduced.

2.1 Software performance engineering

Software engineering aims to deal with the challenges of the software development process by means of an engineering discipline. A characteristic of such is the availability of a catalog of methods and practices plus guidelines for the systematic selection of these practices. The goal is to deliver predictable quality, cost and time-to-market for the engineered product.

A non-functional attribute which is often of high importance during software development is the performance of the developed system. If systems suffer from insufficient performance, they are usually not applicable and cause projects to fail. Therefore, performance prediction of software systems is in the focus of research since the end of the 1990's [9]. The term performance often characterizes the timing behavior and resource efficiency of hard- and software systems. The most important performance properties are response time, throughput and resource utilization [7].

Software Performance Engineering (SPE) is rooted in the research of Connie U. Smith [18]. It aims at evaluating the performance of (software) systems by offering different methods like analytical modeling, simulation, and taking measurements. The SPE's goal is to conduct performance evaluation of software architectures as early as possible [17].

Starting point for most approaches in SPE is the system's software architecture [2, 9]. The architecture can be enhanced with performance annotations. This software model must then be transformed into a performance model (e.g. Queueing Networks, Petri Nets, Markov Chains) and solved for the metrics of interest (see Figure 1).

In the area of SPE, the idea to use model-driven techniques gained attention because they provide automated transformation from source to target performance model(s). However, model-driven performance prediction methods require suitable meta-models like [13, 17, 24]. These meta-models formalize the syntax and semantics of the source and target model. The automatic transformation and processing simplifies the software performance engineering approach and makes it less error-prone.

2.2 The Palladio Component Model

The following describes the Palladio Component Model (PCM), a performance meta-model we used in work to model the performance behavior of the storage virtualization under study. The following gives a brief overview of the PCM. A more detailed and technical description is given in [4].

The PCM is a domain specific meta-model. It describes the performance-relevant aspects of component-based software architecture, often used in business information systems. The PCM approach provides tools (the PCM bench[1]) to create and analyze PCM performance model instances. According to the definition of Lau [11], in this work a *component model* defines what components are, how they can be constructed and represented, how they can be composed or assembled and how they can be deployed.

Software components are units of composition with explicitly defined provided and required interfaces [19]. The kind of components the PCM aims for are entities of business information systems. However, the PCM can be used to model any other entity of a software system. The performance of such a software component is influenced by four factors [9], depicted in Figure 2. The PCM meta-model provides means to take all of these factors into account.

Figure 3 depicts the PCM modeling process, divided into four different views according to these four influence factors. Each influence factor is modeled by a specific role, specifying the performance-influencing factors in separate models. The composition of these models forms a PCM instance.

The *Component Developer* models the components and their implementation. At first, the interfaces (comparable to signature lists) which are provided or required by a component are specified. Provided interfaces specify the services a component offers. Required interfaces are external services the component needs to fulfill its purpose. Furthermore, the component developer models the internal behavior of the provided service(s). To this end, the PCM provides a description language, called *ResourceDemandingServiceEffectSpecification* (RDSEFF), to specify the control flow (e.g. branches, loops, external service calls) and the resource usage of the provided service(s).

[1]http://www.palladio-approach.net/

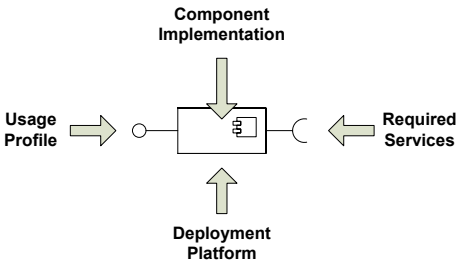


Figure 2: Factors influencing component performance [9].

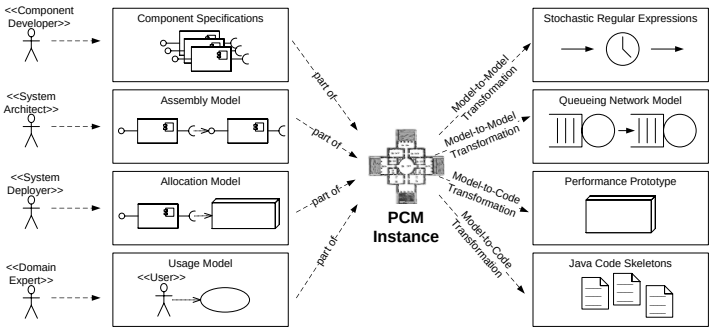


Figure 3: The PCM process [4].

Second, the *System Architect* models the structure of the system. By interconnecting components via their provided and required interfaces, it can be specified which other services a component uses. Hence, the overall system's performance depends on the selection of components, e.g. if a database cache component is used or not.

The *System Deployer* maps the components of the system model to physical resources. Thereby, the influence of the deployment platform on the system performance is specified. Therefore, he must model the hardware environment(s) the system is executed on (like middleware, processor speed, network links, etc.) Furthermore, he allocates the system components onto the specified hardware nodes.

The *Domain Expert* models the performance influence of the usage of the whole system. The Domain Expert describes which parts of the system get invoked by the system's end-users. This description also comprises the kind of workload issued to the system (open or closed). The workload specifies for example how many users invoke the system or the interarrival time of invocations.

Usually, the performance of software does not depend on constant parameter values because e.g. resource demands or system usage may vary during execution. The PCM offers *random variables* to express the uncertainty of such parameters. Random variables can be specified by various probability distributions functions. Furthermore, it is possible to specify mathematical expressions by combining variables with mathematical operators. For further details see [4].

2.3 Reference system

The following describes the basic architecture of IBM systems which are subject to the integration of a storage virtualization layer. In this case study, storage virtualization layer designates an abstraction layer for e.g. direct attached storage devices or storage area networks. Moreover, this section explains how the Virtualization Layer for I/O (VL) could be integrated. Finally, two possible design alternatives for the implementation of the VL are presented. These are a synchronous and an asynchronous implementation, explained later in this section.

2.3.1 System architecture and virtualization layer integration

The core element of the systems under investigation is the hypervisor, depicted in the middle of Figure 4. It virtualizes and separates the hardware (processing resources, main memory, etc.) of an IBM system into several logical partitions (LPAR). Each LPAR can be considered as a separate, smaller instance of the whole system. The hypervisor enables each LPAR to be configured with an individual amount of the system's processing resources, main memory and I/O devices. LPARs can either host a special operating system (e.g. z/OS*) or can again be virtualized (e.g. by z/VM*) to host further guest systems. These guest systems are sharing the hardware resources of the LPAR. Both the single operating systems and the virtualized guests are users of the storage hardware, hence called clients in the remainder.

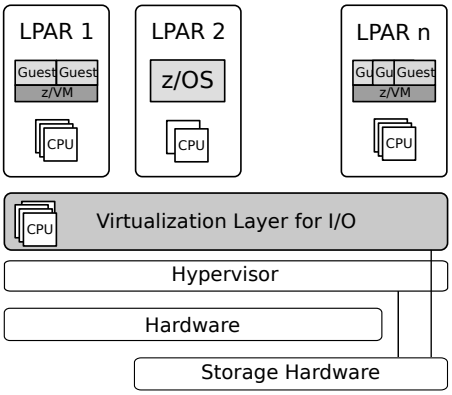


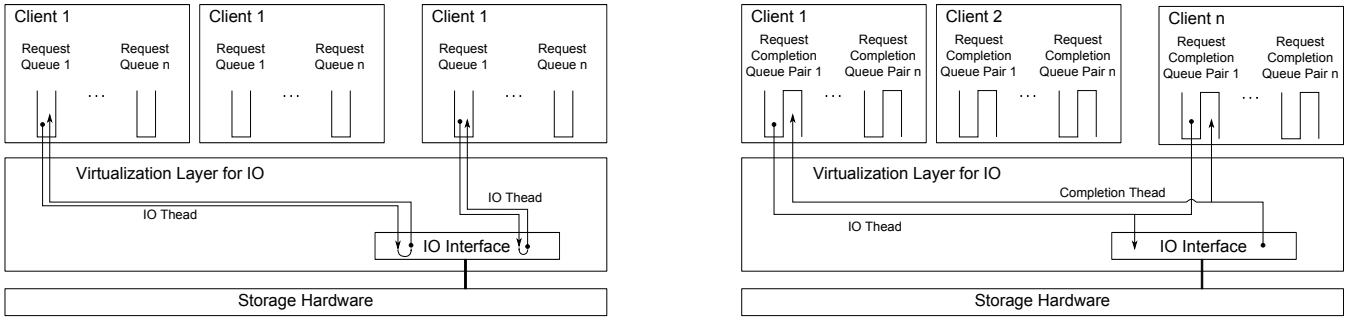
Figure 4: Abstracted system architecture.

Currently, the hypervisor handles all client I/O requests. The approach investigated in this paper is to migrate the storage I/O request processing into a dedicated VL. The VL is privileged to directly access the memory of LPARs. This avoids store and forward of I/O requests. The VL itself is an application running in a specialized operating system. To access the storage hardware, the VL uses the I/O interface provided by the underlying operating system. The I/O interface is connected to the storage hardware by channels. Each channel has a bandwidth (bytes/second) and throughput (requests/second) restriction.

2.3.2 Virtualization layer design

The VL is situated between the request issuing clients and the request processing storage hardware, similar to a message broker between sender and receiver. The request handling can be accomplished e.g. in a synchronous or asynchronous manner. These are the two design alternatives

*Trademarks of IBM in USA and/or other countries



(a) synchronous request handling

(b) asynchonous request handling

Figure 5: Synchronous and asynchronous virtualization layer design alternatives.

investigated in this case study. They basically correspond to synchronous and asynchronous message handling and are described in the following.

The clients have separate *queues* (see Figure 5) for each storage device they are configured with. The queue is a part in the client's memory where requests are stored for processing. The storage device is the receiver of a request. If a client wants to issue a new request to a device, it puts the request in the corresponding request queue. How these stored requests are processed will be explained in the following. The request queues cannot be accessed concurrently by I/O threads. Actually, if a thread accesses a queue, the access might be blocked because the queue is already in use by another I/O thread. However, I/O threads will not block threads from the client's side and vice versa. This can be achieved by a special queue design. In both design alternatives, the I/O interface to the storage hardware is accessible in parallel. The VL's behavior can differ as follows.

Synchronous design: The VL contains a theoretically unlimited amount of *I/O threads*, handling client requests. An I/O thread accesses a client's requests queue, collects a request (if available) and sends the request to the storage hardware by passing it to the I/O interface (see Figure 5a). While the request is executed by the storage hardware, the thread waits for the result and signals it back to the client after completion. Then the I/O thread continues processing the next requests in the queue. If a request queue is empty, the thread accesses one of the remaining request queues and continues with request processing.

Asynchronous design: In this case (see Figure 5b), the forwarding of requests to the storage hardware is decoupled from sending the results back to the clients. Here, a fixed amount of I/O threads works on the request queues. The I/O thread's work is completed after sending the requests to the storage hardware via the I/O interface. Instead of waiting for the result, theI/O thread processes the next request. To receive the results, the clients are equipped with an additional completion queue where the results of requests are signaled to. The signaling is accomplished by another thread type, the *completion thread*. Such a thread is started by the VL as soon as results of the storage hardware become available. It may happen that a completion thread is blocked because another completion thread signals a result to exactly the same queue.

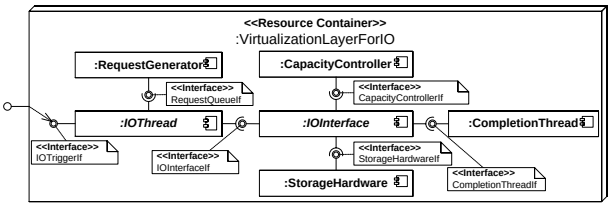


Figure 6: Overview of the functional model.

The differences in the proposed design alternatives regarding their influences on the system's performance lead to questions like the following: Which design alternative provides better performance? What influence has the amount of used threads and queues on the system throughput? How many threads are necessary to handle a specific system load? How does queue blocking influence the overall performance? What VL hardware configuration is required to provide an efficient virtualization? Where are the system's performance bottleneck(s)? The model described in the following shall answer such questions without having to implement each design alternative.

3. MODEL IMPLEMENTATION AND CALIBRATION

The following describes the implementation and calibration of the developed performance model, following the SPE process by Smith explained in Section 2.1. First, we implemented a functional model according to the system and the design alternatives described previously. Second, we conducted several experiments on a prototype to derive the resource usage of each model component. Finally, we used these measurements to calibrate the functional model with realistic resource demands.

3.1 Functional model description

The developed PCM model instance is depicted in Figure 6. Its illustration follows a simplified combination of the UML deployment diagram and the UML composite structure diagram. The following explains the modeled components according to the control flow through the model.

The control flow of the VL starts at the system interface depicted on the left. A client invokes the system via this interface. This invocation is delegated to the *IOThread*

component. Now the *IOThread* calls the *RequestGenerator* component to get a request in return. The *RequestGenerator* component represents the clients and their request queues. It can be configured by several parameters like request size, request type (READ or WRITE), amount of queues, etc. The *IOThread* demands a specific amount of CPU time before passing the received request to the *IOInterface*. The *IOInterface* calls the *CapacityController* used to assure the throughput constraint of the I/O interface. In case the maximum throughput of requests is reached, further requests will be delayed. Afterwards, the *IOInterface* consumes some CPU time and forwards the request to the *StorageHardware*. This component models the request execution on the physical hardware by a delay depending on the request's size and type. Subsequently, in the synchronous case, the control flow returns to the client. In the asynchronous case, the control flow already returned from the *IOThread* to the user, as the call of the *IOInterface* was forked. Hence, in the asynchronous case the *IOInterface* calls the *CompletionThread*, which signals the results and thereby the end of the transaction to the client. The *CompletionThread* also requires a specific amount of CPU time.

The behavior of the *IOThread* and *IOInterface* differs in the synchronous and asynchronous case. Therefore, the performance model includes a synchronous and asynchronous version of these components. This offers the flexibility to easily switch between the different design alternatives.

Modeling restrictions: As already mentioned, the target domain of the PCM is business information systems. Therefore, the model could not be created as straight forward as it might appear. Several work-arounds were required to implement a performance model equivalent to the system. One problem was that components have no active behavior. They must be invoked by either a client or other components. The modeling of the I/O thread in the virtualization system requires such active behavior. To realize this we extended the *IOThread* component with a trigger interface. This interface is then called from an external workload.

Second, currently the PCM does not support automated replication of components. However, this is required to explicitly model a varying (one to 100) amount of request queues. The *RequestGenerator* solves this issue. It is an additional component representing all request queues, reflecting the behavior of queue accesses and returning requests. Depending on the parameter settings of the *RequestGenerator*, the *IOThread*'s call is delayed to simulate blocked queue accesses.

Third, the current version of the PCM has no direct support to model component state. Hence, it is impossible to count and limit the throughput of a component directly. The *CapacityController* component encapsulates and reflects the throughput restriction by using the features PCM currently provides (forks, passive resources and delays). A more detailed description of the model and the specific behavior implementation of each component is given in [6].

3.2 Experiment setup

The following describes the experiment setup to determine the model parameters. The calibration and validation of the functional performance model is based on the experiment-based derivation of software performance models presented in [5]. This approach is inspired by the general ideas and rules proposed by Jain [7]. It combines existing knowledge of

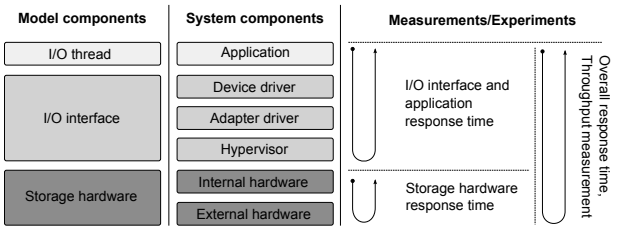


Figure 7: Measurement setup architecture.

the system under study with iterative, goal-oriented experiments. These experiments support performance analysts in identifying valid assumptions for performance modeling. They help to assess the prediction accuracy of the model. Furthermore, it is important that the performance model design is driven by a specific goal. This directs the design effort to the factors of interest, similar to the GQM-approach [3].

All experiments and measurements were executed on a System z9 with 48 processors and 128 Gigabyte of main memory. The storage controller was a DS8000*, connected via four 8 Gbit/s FCP (Fiber Channel Protocol) channels.

In the experiments, two different variables were observed. Response time measurements were conducted for calibrating the resource demands of the model components. Moreover, we measured the throughput used to validate the model described in Section 4. In the following, the system's throughput (X) is defined as the ratio of requests (R) per time (T), i.e. $X = R/T$.

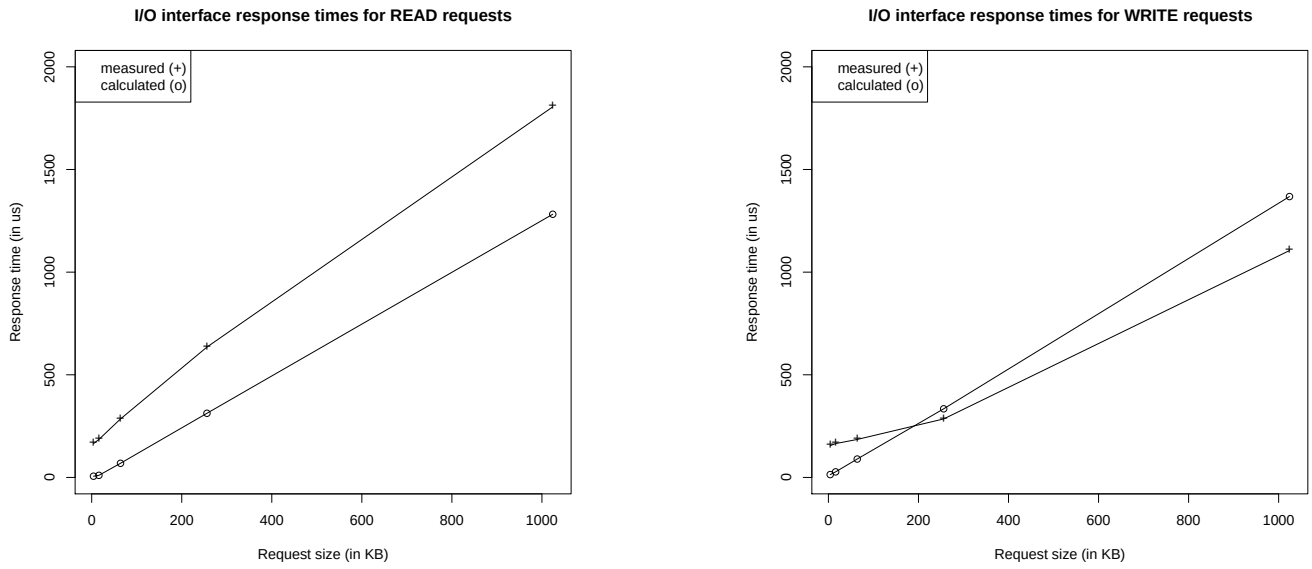
The measurement data was gathered with two different tools. One to identify the response times of different components of the request handling stack depicted in Figure 7. The second tool generates system load and measures the overall system throughput and response time.

The generated workload was a closed workload with a configurable amount of clients, issuing request to the system. Open workloads were not supported by IBM's load generator. In our experiments, we measured the throughput for a varying amount of clients (1,2,4,..., 256) and different request sizes (4KB, 16KB, 64KB, 256KB, 1024KB) and types (READ, WRITE). Moreover, we ascertained response times for the same request size/type combinations with an additional tool. However, for this measurements we restricted the amount of clients to one client to avoid mutual disturbances. The results are listed in Table 1. In this approach we chose a synthetic benchmark to calibrate the model with reasonable costs for measurements. However, the benchmark was designed to cover different load conditions and request sizes. Nevertheless, in future work, we plan to simulate and analyze the model with realistic workloads.

3.3 Model calibration

The response time measurements of the experiments are used to derive the resource demands of the functional performance model. However, the use of the additional tool to measure the response times leads to a throughput lower than without the tool. This indicates a disturbing influence of the response time measurement on the throughput characteristic of the system. If the measured response times are used to configure the model, the measured throughput (with re-

*Trademarks of IBM in USA and/or other countries



(a) (b)

Figure 8: Measured (+) and calculated (o) I/O interface response time for READ (Fig. 8a) and WRITE (Fig. 8b).

READ	**4KB**	**16KB**	**64KB**	**256KB**	**1024KB**
I/O interface + I/O thread (µs)	180	200	300	650	1820
Storage hardware (µs)	100	160	420	1490	5160
Overall response time (µs)	270	360	720	2140	6980
Throughput (req./sec.)	**4KB**	**16KB**	**64KB**	**256KB**	**1024KB**
with response time measurement	3600	2750	1400	470	145
w/o response time measurement	6219	4707	2184	656	223
WRITE	**4KB**	**16KB**	**64KB**	**256KB**	**1024KB**
I/O interface + I/O thread (µs)	170	180	200	300	1120
Storage hardware (µs)	250	380	890	2180	5830
Overall response time (µs)	420	560	1090	2480	6950
Throughput (req./sec.)	**4KB**	**16KB**	**64KB**	**256KB**	**1024KB**
with response time measurement	2350	1780	915	400	145
w/o response time measurement	3276	2286	1130	498	175

Table 1: I/O Interface + I/O thread, storage hardware and overall response times, and system throughput with and without simultaneous response time measurement.

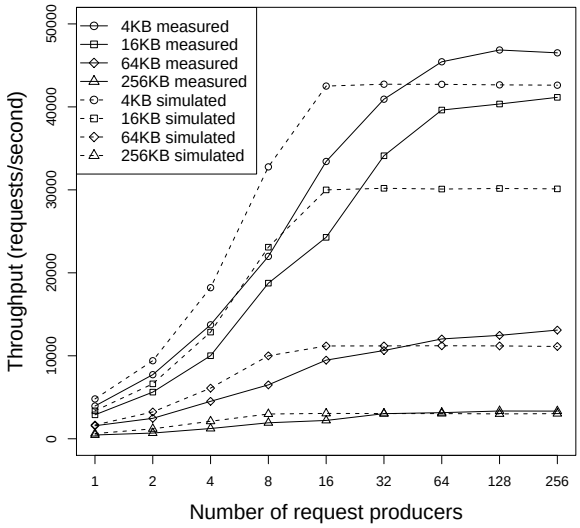
Request		**Initial Throughput**			**Maximum Throughput**		
Type	Size	Measured	Simulated	Rel. Error	Measured	Simulated	Rel. Error
READ	4	6219	6049	2.73%	47416	48706	2.72%
	16	4707	4539	3.57%	39518	37879	4.15%
	32	3580	3372	5.81%	22530	23196	2.96%
	64	2184	2183	0.05%	11911	12889	8.21%
	256	656	688	4.88%	3053	3125	2.36%
	1024	223	184	17.49%	771	820	6.36%
WRITE	4	3276	3628	10.74%	34487	36355	5.42%
	16	2286	2384	4.29%	23700	22906	3.35%
	32	1690	1636	3.20%	15574	15338	1.52%
	64	1130	1221	8.05%	9553	9144	4.28%
	256	498	543	9.04%	2863	2593	9.43%
	1024	175	168	4.00%	723	717	0.83%

Table 2: Absolute and relative errors for initial and maximum throughput of measurements and simulation for the final configured model.

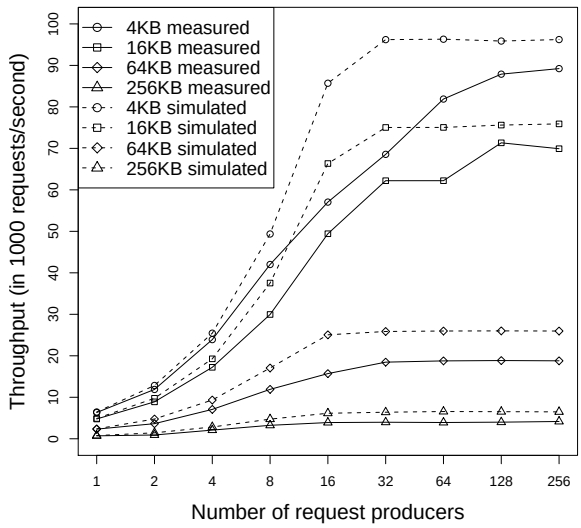
sponse time measurement tool) is predicted precisely. However, the performance model should predict the throughput without reflecting the influences of the response time measurements. Hence, the actual resource demands could not be calculated directly from the response time measurements. We could only use the trends discernible in the measurement data and the ratio of different response times to parameterize the model components.

The I/O thread's runtime is independent of the request type or size, as the thread simply passes the memory address to the I/O interface and vice versa. In contrast, the I/O interface's resource usage depends on the request's size and type. Several tests with a temporary configured model revealed that the I/O interface response time significantly influences the throughput, especially for high system load. Hence, the measured I/O interface response time cannot lead to an accurate throughput prediction because the measured response time is simply too long. This can be understood with the following example calculation. If a throughput of e.g. 6219 requests per second for a 4KB READ request is achieved, the overall response time is at most $\frac{1}{6129} = 160\mu s$ for one request. Hence, we calculated the I/O interface response time from the throughput measured without the additional response time measurement tool and used the trends depicted in Figure 8 for the correlation of request size and response time. This is described in more detail in [6].

After this calibration, the predicted throughput deviation was below 10% for most of the measurements. The average error over all data was 5% (see Table 2). *Initial Throughput* lists the throughput measurements for one request issuing client. This throughput is only influenced by the hardware response time and hence is proportional to the system response time. *Maximum Throughput* denotes the throughput measured and predicted for 256 request issuing clients.



(a) Throughput comparison for READ/WRITE mixtures



(b) Throughput comparison for two CPUs

Figure 9: Model validation by throughput comparison for two scenarios.

4. EVALUATION

To validate the created performance model and to evaluate the simulation results we used a two-step approach. The first step validates the synchronous performance model by comparing the simulation results with measured experiment results conducted on the prototypical synchronous design alternative.

In the second step we compare the results of the validated synchronous model with the asynchronous model. Note that we use only one single model instance which is calibrated and parameterized once and have no separate models for the synchronous and asynchronous design alternatives. The design alternatives can be simulated by simply replacing components, which has no effect on the calibrated resource demands of the model.

4.1 Model validation

To validate the performance model, we conducted two specific throughput measurement series and compared the results to the simulated throughput. In one series, the request type (a mix of 60% READ and 40% WRITE requests) differs. In a second series, we used the same mix and added an additional CPU to the VL. For both series, we measured the throughput for 1,2,4,...,256 clients (see Figure 9).

The diagrams show a qualitatively high correlation of measured and predicted throughput. The predictions are not completely accurate, but tend to the same error behavior, e.g. the initial throughput (1 client) is overestimated whereas the maximum throughput (256 clients) is underestimated for all sizes of the plain READ/WRITE mix. On average, the relative prediction error $f = (x_{sim}/x_{meas}) - 1$ for all measurement data for the READ/WRITE mix is below 19% and below 21% for the mix with an additional CPU.

However, predictions show a relatively high quantitative discrepancy in some cases, e.g. the 16KB READ/WRITE mix, which leads to further investigations. The discrepancy can have its origin in the measurements. For example, although we configured the tool to avoid caches, cache hits cannot be completely precluded, especially in case of the READ/WRITE mix. However, each cache hit can cause a considerable speedup in the measured throughput. Another reason could be an absent detail in the performance model, e.g. a possible influence of a scheduling overhead for two CPUs. To test this assumption, we provisionally integrated a more detailed scheduler simulation for multicore platforms [5] into the PCM. This scheduler improved the prediction accuracy for the maximum throughput, in which case the CPU is the bottleneck, by about 7% on average.

Additional measurements would have been necessary to obtain further system details and to create a more accurate model. However, one must trade-off the benefits of a more accurate model with the cost and effort for creating the model. Therefore, despite the quantitative errors the qualitative prediction accuracy of the model was considered to be sufficient to discern the system behavior as the tends are accurately simulated.

4.2 Discussion of simulation results

The validated model can now be used to vary parameters like request size, request type, amount of threads and queues, and design alternatives like synchronous and asynchronous components to observe their influences on the system performance. As there exists no asynchronous system prototype, the validated parameters of the synchronous model were used to parameterize the asynchronous model, too. This is feasible as the design alternatives only vary in the behavior of the I/O thread and the completion thread, respectively, not in the resource demands of the I/O interface or storage hardware components. To simulate the asynchronous performance model, the synchronous components must simply be replaced by their asynchronous counterparts.

The comparison of the simulation results of both design alternatives unveils little differences w.r.t. the selected metric throughput. Also the response times in both scenarios are very similar for closed workloads. However, a crucial difference is observable if an open workload with an exponentially distributed interarrival time is used instead of a closed workload. In this case, the asynchronous design alternative is more capable in handling peak loads. In the synchronous case, the response times were distributed relatively constant compared to the closed workload whereas they improved in the asynchronous case (see Figure 10). The open workload also demonstrates that one to ten asynchronous I/O threads are capable of handling the same load as synchronous I/O threads. Additional asynchronous I/O threads do not further improve the results.

In the synchronous case, there is one active I/O thread per request. However, in the asynchronous case, more than n I/O threads are active. In addition, there is one thread per request for handling the request within the I/O interface. Furthermore, completion threads are required to signal the results. Nevertheless, the model revealed that the influence of the amount of I/O threads and completion threads on the throughput is negligible because their runtime is insignificant compared to I/O interface and storage hardware response times.

Several conclusions can be drawn based on a comparison of both models. The differences between both implementations concerning the system throughput is low. Hence, one must consider the advantages and drawbacks of the design alternatives itself. For example, the synchronous version is easier to implement and to maintain in case of malfunctions. Moreover, it has an intrinsic overload protection as the synchronous threads must wait for the result and cannot send more requests to the I/O interface as the I/O interface can process. However, the asynchronous implementation offers higher flexibility and better responsiveness in case of peak loads. Hence, in this case study the decision whether to use a synchronous or asynchronous approach is mainly depending on other factors than performance.

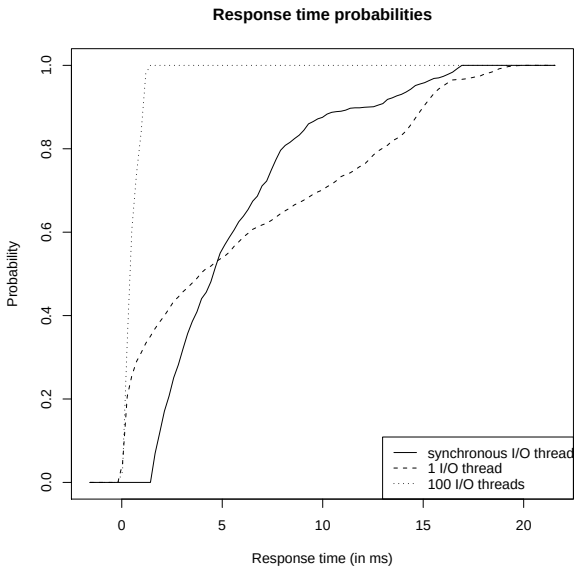


Figure 10: Cumulative distribution function of the response time for 1 and 100 asynchronous I/O threads and the synchronous I/O threads.

5. EXPERIENCES GAINED

Software performance engineering is usually motivated by cost savings that are achieved by detecting performance issues in an early phase of the development process [9, 17, 23]. Moreover, models promise to support system understanding and improve the system development. In this work we collected experiences which support these statements and learned how to use models properly. Furthermore, shortcomings of the used meta-model could be identified, useful for further improvements of the PCM.

Performance models must be elaborated thoroughly and the creation of performance models can cause high initial costs. Nevertheless, this work demonstrates that performance models provide a quick, easy, and flexible way to compare and analyze design alternatives without implementing prototypical design alternatives. Hence, they provide a valuable alternative to performance prototypes or performance measurements in real systems. For example, the PCM model offers the flexibility to easily switch between the modeled design alternatives and vary their parameter settings to observe the influences on the performance. This shows another lesson learned. Performance models are abstractions of systems. Hence, the performance model can concentrate on relevant and important parts of a system. Especially for complex systems like the System z9, the focussing on relevant factors and their easy changeability offers a higher flexibility in system analysis.

However, one must consider that there is always a tradeoff between model accuracy and modeling effort. For example, in this case study the prediction results revealed certain inaccuracy. The reason for this could have been identified with additional, more costly measurements. However, the objectives were already achieved with the presented model ("Make everything as simple as possible, but not simpler!").

The case study shows that performance modeling actually works and that a creation of a performance model can replace further prototypical implementations. We estimate the effort of creating a calibrated and validated performance model for an unexperienced person which is not familiarized with the system and the PCM to be about four person months. The effort to implement a plain asynchronous VL running in an LPAR is at most three person months. However, one would still need to implement a simple and flexible configuration of the VL and the communication with other partitions. Already available at IBM were the front-ends to generate traffic and the back-end drivers. If all this had to be implemented and taking into consideration all the required skills, the effort to implement a full-fledged prototype would be at least 24 people months. Hence, in a scenario like this where knowledge of a complex system is distributed over several departments, a performance model can be created by few people with much less effort than a performance prototype. This fortifies that the initial effort for creating a performance model is high but if a model is available, it is an easy, cheap, and flexible way to investigate different alternatives.

Furthermore, the performance model has identified other performance bottlenecks than originally expected. It revealed that the influences of the amount of threads and

queues on blocked queue accesses on the system performance are eclipsed by the *IOInterface* and the *StorageHardware*. Besides, the model demonstrates that optimizing the storage hardware influences the initial throughput of the system, i.e. if system load is little, faster hardware can improve the throughput. In contrast, to increase the throughput for high system load, it is necessary to improve the I/O interface response time by decreasing its CPU resource demands.

We experienced ourselves that performance models support the design and analysis, and improve the understanding of existing systems. During the modeling and calibration, the simulation results caused revisions of measurements results and assumptions about the system behavior. For example, the model indicated a higher throughput for two CPUs than the measurements. The flaw was the bandwidth limit which was hit by the measurements and could be fixed by adding an additional channel.

Concerning the applicability of the PCM, we learned that this performance meta-model is practical and capable of modeling component-based software architectures. Furthermore, the created model was able to predict the performance behavior with sufficient accuracy. However, to model software and system components with more detail (e.g. queues and threads), the current features of the PCM are insufficient. Work-arounds had to be implemented to model circumstances like queue blocking or throughput constraint. Hence, the PCM requires improvements for better applicability outside the domain of business information systems. The performance metrics and the visualizations the PCM offers were sufficient for the purpose of this case study. They depicted resource utilization and response times of the system and components very detailed. However, a more flexible database to manage simulation runs and their results would be a great benefit.

By means of a real system, this industrial case study shows that model-driven performance engineering works, even in a virtualized environment. No experts for performance models are required, as by tool support performance models can be created and easily modified on a very high level of abstraction.

6. RELATED WORK

There are two different areas of research this work is related to. Distantly related is the performance analysis of I/O virtualization and closely related performance modeling, especially case studies about methods and tools.

Concerning I/O virtualization, there exist at least two approaches as examined in [20, 21]. Wiegert et al. analyze the performance impact of improving the internal setup (*scale-up*) of an I/O virtual machine monitor (IOVM) on scalable networking [21]. In several experiments the authors examine different configurations of cores, context distributions and thread types. In particular, results show that moving from a single processor to a SMP configuration further improves throughput. Wei et al. propose and evaluate solutions for delivering scalable network performance on a multi-core platform [20]. The authors want to achieve a performance increase by moving the I/O virtualization work out of the hypervisor onto dedicated IOVMs (*scale-out*). Their experiment results and performance comparisons show improved efficiency and flexibility of dedicated IOVMs compared to a centralized solution.

The paper of Ramesh and Perros describes a multi-layer client-server queuing network model with synchronous and asynchronous messages [16]. Their work is motivated by CORBA, where distributed objects use the client-server interface to communicate by synchronous and asynchronous messages. Although motivated by practical topic, the model is very generic as the focus is more on model analysis then performance modeling. The authors analyze the model for one-layer and multi-layer networks and compare the simulation results with performance measures to evaluate the accuracy with good results.

The Proactor/Reactor patterns used in middleware are closely related to the synchronous and asynchronous I/O request handling analyzed in this work. The authors of [14] present a Queueing Model which captures the performance relevant characteristics of this Proactor pattern. They analyze the performance of a Proactor-based Web server, concentrating on metrics like throughput and response time. In a second paper, the authors present a performance model of the Reactor pattern based on Stochastic Reward Nets (SRN) [15]. Their analyses are the same as in the previous paper and they evaluate the same performance metrics as response time, throughput and loss probability. Again, a case study illustrates the use of the model. However, yet there is no comparison of the results.

Another related approach uses Queueing Petri Nets to model and evaluate a deployment of the industry-standard SPECjAppServer2004 benchmark for J2EE application servers [8] . Moreover, it explains a practical performance modeling methodology to construct accurate performance models. Results show that QPNs are capable to predict the performance of distributed component-based systems systems.

An evaluation of different approaches of model-based performance prediction of component-based systems is given in [2] and [9]. The major benefit of the component-oriented PCM is its flexibility. It enables easy exchange of components (e.g. synchronous with asynchronous implemented components) to observe the model behavior. For the PCM, there are two studies examining the applicability of the PCM in a theoretical [10] and industrial context [1]. The results present the PCM as a practical tool for modeling business information systems but did not investigate the applicability besides business information systems. Additionally, the applicability of the PCM approach was investigated in an experiment [12]. The results show that the quality of the models and predictions created by the test subjects deviated less than 10% from the predictions achieved with a reference model created by the experimentators. Furthermore, over 80% of the subjects were able to rank the given design alternatives correctly, which indicates the appropriateness of the approach itself.

7. CONCLUSIONS AND OUTLOOK

This work is an industrial case study whether software performance engineering, particularly the PCM approach, can be applied in industry. For IBM this work is an investigation if the PCM is suitable for their requirements. Although other approaches (e.g. QPNs) might appear more suited, the PCM with its mature toolset was intentionally chosen as the preferred performance model to test its applicability in a domain outside of business information systems.

The resulting model was used to analyze two virtualization layer design alternatives, investigated as a proof of concept on a System z9. The analysis results revealed that the

two examined design alternatives (synchronous and asynchronous request handling) have no difference concerning the metric throughput of the system, even if the workload type is varied. However, the asynchronous implementation has slightly better response times for peak loads and is able to handle the same amount of system load with fewer threads. The advantages of a synchronous approach is an easier implementation with intrinsic overload protection.

Moreover, this work shows that performance models help to get a performance abstraction of a system in a quick and cost effective way because no alternative prototypes must be implemented. Furthermore, it demonstrates that performance models can be created without the need to involve all system experts required to create a performance prototype. Simplified, one person with sufficient system knowledge is able to implement an accurate performance model which achieves convincing prediction results.

However, the PCM has some shortcomings (e.g. no component state, no component replication) which must be resolved in future work to better support scenarios outside the domain of component-based software architectures. Nevertheless, the PCM can be considered as a mature approach with a tool which is capable to describe a system's performance abstraction and predict a system's performance behavior in an efficient way.

Future work will conduct further measurements to refine and improve the knowledge and the model of the examined system. Furthermore, the identification and usage of realistic workload profiles can improve the prediction quality of the model and even lead to a conclusion which implementation fits best for real workload profiles. Additionally, with a classification of workload profiles, one could think of using a performance model during deployment to calibrate a system to the customer-specific workload. Furthermore, a quantitative cost-benefit comparison of a modeling approach with the PCM and a performance prototype implementation is important to get quantifiable statements about the effectiveness of a performance modeling approach. Finally, a comparison of the PCM with other performance models and tools like QPNs in terms of effectiveness would be of interest.

8. REFERENCES

[1] R. Andrej. Evaluation of the prediction approach "Palladio" in the industrial context of the CAS Software AG, 2008. Master's thesis (in German).

[2] S. Balsamo, A. D. 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] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[4] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Sys. and Softw.*, 82:3–22, 2009.

[5] J. Happe. *Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments*. PhD thesis, Univ. of Oldenburg, 2008.

[6] N. Huber. Performance Modeling of Storage Virtualization. Master's thesis, University of Karlsruhe (TH), 2009.

[7] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling*. John Wiley & Sons, April 1991.

[8] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, July 2006.

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

[10] K. Krogmann and R. H. Reussner. *The Common Component Modeling Example*, volume 5153 of *LNCS*, pages 297–326. Springer-Verlag Berlin Heidelberg, '08.

[11] K.-K. Lau. Software Component Models. In *ICSE'06*, pages 1081–1082. ACM Press, 2006.

[12] A. Martens, S. Becker, H. Koziolek, and R. Reussner. An Empirical Investigation of the Applicability of a Component-Based Performance Prediction Method. In *EPEW'08*, volume 5261 of *LNCS*, pages 17–31. Springer-Verlag Berlin Heidelberg, 2008.

[13] Object Management Group (OMG). UML profile for schedulability, performance and time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02.

[14] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray. Performance analysis of an asynchronous web server. *COMPSAC '06*, 2006.

[15] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray. Performance analysis of a middleware demultiplexing pattern. In *HICSS '07*, 2007.

[16] S. Ramesh and H. G. Perros. A multilayer client-server queueing network model with synchronous and asynchronous messages. *IEEE Trans. Softw. Eng.*, 26(11):1086–1100, 2000.

[17] C. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley, 2002.

[18] C. U. Smith. Increasing information systems productivity by software performance engineering. In *Int. CMG Conference*, pages 5–14, 1981.

[19] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2002.

[20] J. Wei, J. R. Jackson, and J. A. Wiegert. Towards Scalable and High Performance I/O virtualization - A Case Study. In *HPCC '07*, pages 586–598, 2007.

[21] J. Wiegert, G. Regnier, and J. Jackson. Challenges for scalable networking in a virtualized server. *ICCCN '07*, pages 179–184, 2007.

[22] L. G. Williams and C. U. Smith. Making the business case for software performance engineering. In *29th Int. CMG Conference*, pages 349–358, 2003.

[23] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE '07*, pages 171–187, 2007.

[24] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (puma). In *WOSP '05*, 2005.