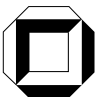


Universität Karlsruhe (TH)
Research University · founded 1825

Performance modeling of storage virtualization

Diploma Thesis at the
Institute for Program Structures and Data Organization
Chair for Software Design and Quality
Prof. Dr. Ralf H. Reussner
Fakultät für Informatik
Universität Karlsruhe (TH)

by

cand. inform.
Nikolaus Huber

Advisor:
Prof. Dr. Ralf H. Reussner
Dipl.-Inform. Christoph Rathfelder

Date of Registration: 2008-11-01
Date of Submission: 2009-04-30

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2008-04-30

Virtualisierung ermöglicht die gemeinsame Nutzung von Hardware und spielt deshalb bei der Serverkonsolidierung eine wichtige Rolle, weil Kosten für Platz und Management eingespart werden können. Einer der Teilbereiche von Virtualisierung ist die I/O Virtualisierung, welche z.B. Speicherinfrastruktur zur gemeinsamen Nutzung bereitstellt. Häufig sind bei dem Entwurf und der Implementierung von Systemen die Einflüsse verschiedener Parameter oder von Entwurfsentscheidung auf die Leistungsfähigkeit des Systems unbekannt oder nur schwer einzuschätzen. Performance-Modelle bieten jedoch die Möglichkeit, das Verhalten von Systemen zu modellieren um es anschließend zu analysieren und zu bewerten. Das Palladio Component Model (PCM) ist eine Modellierungssprache zur Beschreibung von komponentenbasierten Softwarearchitekturen. In dieser Arbeit wird mit Hilfe von PCM ein Modell für verschiedene Entwurfsalternativen einer Virtualisierung der Speicherinfrastruktur entwickelt. Die Kalibrierung und Validierung des Modells erfolgt anhand von Messdaten eines bestehenden Prototyps. Aufbauend auf diesem Model werden zum einen die Auswirkungen verschiedener performance-relevanter Faktoren auf die Leistungsfähigkeit untersucht und zum anderen Entwurfsalternativen verglichen. Darüberhinaus ist diese Arbeit eine Fallstudie für die Anwendbarkeit von PCM und dessen Fähigkeiten im industriellen Kontext außerhalb seiner Domäne der komponentenbasierten Softwarearchitekturen.

Contents

1. Introduction

Virtualization techniques offer the possibility to use highly available and efficient computer systems to consolidate a multiplicity of serves on one single machine. This consolidation can reduce the cost and management overhead [RG05]. An example for such big virtualized hardware is the System z[1], allowing to run hundreds to thousands of servers on a single mainframe. This need for big, virtualized hardware requires improved and efficient virtualization solutions. The focus of current research is not only on CPU and memory virtualization, but also on I/O virtualization. Having plenty of different storage hardware types of different vendors, an efficient design and implementation of a storage virtualization set an ambitious goal for system developers.

To assess the impact of design decisions on the performance of a system and to evaluate implementation details, models for performance prediction can be supportive. The goal of this work is to implement such a performance model for a potential storage virtualization for an IBM system. This shall be achieved by making performance relevant abstractions of the architecture description by creating a performance model which reflects the behavior of the real system. The model can then be used to analyze and evaluate the performance impact of changing design decisions or varying performance influencing parameters in a quick, easy and cost efficient way.

There are research projects dealing with I/O virtualization, focusing on the performance analysis in comparison to existing systems [WJW07, WRJ07]. However, none of them implements a performance model. The performance modeling approach used in this work is the Palladio Component Model (PCM), designed and developed to analyze the performance of component-based software architectures. Current projects are concentrated on evaluating PCM's applicability in PCM's target domain [KR08, And08], but neither of them explores its applicability in a PCM-foreign domain.

PCM and a performance modeling approach in general has several advantages like cost efficiency or design-time performance analysis. Furthermore, a PCM instance offers flexibility, i.e. it is possible to easily exchange model components by others

[1] IBM System z, http://www-03.ibm.com/systems/z/

with different behavior. This allows to simulate the PCM instance with different configurations to observe their influences on the performance and to compare design alternatives. Moreover, a performance model offers the flexibility to vary several parameters to observe the influences on the system's performance and the system's behavior. These observations are supported by the visualization features of PCM.

However, to analyze the performance influences of different design alternatives of storage virtualization, an appropriate model is required. This model shall provide an easy variation and evaluation of the performance factors.

1.1 Goal

The main goal of this thesis is to create of a performance model for a potential virtualization layer for I/O (VL) for IBM systems, investigated as a proof of concept and implemented on a System z as a showcase. The target is to extract, analyze and model the performance relevant factors of the potential storage virtualization layer so the created model reflects the behavior of a system prototype. Therefore, methodical experiments to derive software performance models will be conducted which evaluate the performance influences of the system. These resulting measurements guide the calibration and validation of the model.

The design alternatives explicitly evaluated in this work are synchronous versus asynchronous VL implementation. Based on the synchronous model calibration, an asynchronous model alternative is created to evaluate this alternative without the need to implement a dedicated prototype. The evaluation covers different parameter configurations of the asynchronous model as well as a comparison of both design alternatives.

Because the modeled system is not located within the target domain of PCM, the whole modeling process itself is a study of PCM's applicability in other domains besides component-based software architectures and business information systems. For problems and difficulties arising from the deviation of PCM's domain, preferably generic solution patterns shall be presented. Hence, this work shall assess the practicability of PCM in a concrete industrial but research- and innovation-affected context and reveal the potential and shortcomings of PCM.

1.2 Outline of the Thesis

This thesis is structured as follows. Chapter 2 discusses the foundations of this work, introducing current virtualization techniques in general and I/O virtualization in particular. Furthermore, the System z architecture is explained, followed by an introduction to PCM. Moreover, a methodology to derive performance models is presented.

In chapter 3, other research related to this work is discussed, starting with performance analysis of I/O virtualization and followed by other projects using PCM to create performance models. As there currently is no research project modeling the performance of I/O virtualization, other approaches to create performance models of systems and the Proactor/Reactor patterns are presented.

Chapter 4 explains the architecture of the system to be modeled. Chapter 5 presents the model created according to the introduced architecture and discusses the challenges and limitations of the modeling process. After that, chapter 6 explains the calibration and validation of the created synchronous performance model.

Chapter 7 is an evaluation of the asynchronous model and a comparison of the two design alternatives under different configurations. Finally, chapter 8 summarizes this work and discusses the advantages and difficulties of the modeling approach. Moreover, it gives an outlook on future work.

2. Foundations

This chapter explains the background of this work and the foundations for this modeling approach. The first and second section deal with virtualization concepts in general and I/O virtualization in particular. The third section gives an overview of the System z architecture and how I/O virtualization could be integrated. In the fourth section, the Palladio Component Model (PCM) as an approach for performance modeling and prediction is introduced. Finally, an approach to derive and validate software performance models based on experiments is presented.

2.1 Virtualization in general

There exist different types of virtualization. Sometimes, terms related to virtualization are used with a different meaning. In the following, virtualization always refers to *full virtualization* (or native virtualization) which means that a machine's hardware like CPU, memory and I/O devices are entirely simulated. This is in contrast to the *emulation* (or non-native virtualization) which allows software applications or operating systems (OS) written for a special type of computer processor architecture to be executed on a different platform.

The first time virtualization was used in the 1960s. At that time, hardware was very large and expensive, especially mainframes. Virtualization provided a convenient way to multiplex that scarce hardware resources. During the 1980s and 1990s, hardware became cheap and mainframes were replaced by minicomputers and later PCs, and virtualization disappeared to the extent that computer architectures no longer provided the necessary hardware to implement them efficiently [RG05]. Today, less expensive and more powerful hardware has led to a proliferation of machines, but these machines are often not fully utilized and require a lot of space and management overhead. This is why virtualization regains importance, especially in server consolidation.

The core of each virtualization concept is the *virtual machine monitor* (VMM, also called hypervisor). Basically, a VMM is an abstraction layer added on top of the bare hardware [Men05] that provides an uniform interface to access the hardware below (see figure 2.1). A *virtual machine* (VM) is the environment created by a VMM

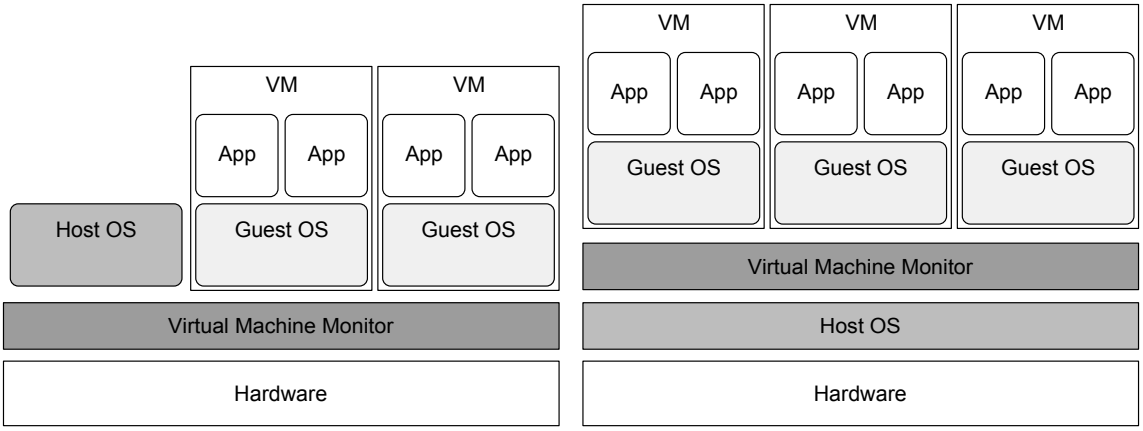


Figure 2.1: The Virtual Machine Monitor as an abstraction layer between hardware and VM (type-I) and between Host OS and VM (type-II).

which is similar to the original machine, but usually with different or shortened hardware resource configurations (e.g. less memory). The following explains the advantages of virtualization and gives an overview of its types and properties. This can be referred in more detail in [RG05, Men05].

As already mentioned, virtualization decouples the software from the hardware (see figure 2.1). This complete abstraction from the underlying hardware enables the VMM to control the hardware access and hardware resource usage of the *guest operating systems*, the operating systems running inside of virtual machines. Another property is that a VMM provides a *uniform view* of the underlying hardware. For example, even if a different I/O system of a different vendor is used, it looks the same for the virtual machine. Hence, administrators can easily exchange the hardware the virtual machines run on.

Vallee et. al. distinguish between two different types of virtualization [VNOS08]. The first is *type-I virtualization* where the VMM runs on the physical hardware, directly. In this case, the VMM is often hosted inside a specialized Linux kernel. An example for type-I is Xen[1]. In contrast, if the VMM runs on a host operating system, it is called *type-II virtualization*, like the VMware server[2] (see figure 2.1).

Virtualization allows to run many different VMs on one single physical machine. Each VM is an environment completely separated from the others, can host a different operating system and can easily be ported onto other machines. Hence, virtualization can solve mobility and security problems by hardware abstraction, hardware multiplexing and separation from other systems. For example in high performance computing (HPC), the need for I/O virtualization and virtualization in general is increasing because it has the advantage that developers can implement new software inside the VM of a local machine which can then easily be ported to a cluster. Moreover, as all systems can be consolidated on one physical machine, virtualization can overcome manageability difficulties e.g. when updating hardware. Furthermore, virtualization provides a possibility for deploying existing and legacy-like software systems on old hardware and an easy way to test the software based on innovative operating systems.

[1]The Xen hypervisor, http://www.xen.org/
[2]VMware, http://www.vmware.com/

Moreover, as the software state of a VM is completely encapsulated, it is easier for the VMM to map or remap available hardware to virtual machines or even to migrate them to a different system. This simplifies load balancing, dealing with hardware failures and eases system scaling.

Concerning security, with virtualization it is possible to distribute applications which previously ran on one single OS over several virtual machines. If an attacker compromises one application executed within one VM, only this VM is compromised and other virtual machines stay unaffected [RG05]. Or if one application crashes the OS, the other applications continue their work. In short, virtualization gains more and more importance, especially in the area of server consolidation and utility computing. Mainly its capability to consolidate several systems on one physical machine - which is the superior motivation of this work - makes it attractive as a cost saver.

Concluded, the VMM must provide a hardware interface for the virtual machines. Various techniques with different advantages and drawbacks can achieve this. The following briefly explains the different types of hardware resource virtualization, whereas section 2.2 explains I/O virtualization in more detail.

2.1.1 CPU virtualization

A CPU architecture is virtualizable if it supports the execution of VMs on the real hardware, while the VMM still has the control of the CPU [RG05]. This is called *direct execution*. In this case, the VMM provides an abstraction of the underlying machine's hardware and transparent hardware access to all VMs. This implies that software, e.g. the operating system can be executed without changes or adjustments. An example for such a VMM is the z/VM hypervisor, designed to run hundreds to thousands of servers on a single mainframe [ea07]. Unfortunately, not all architectures were designed to be virtualizable, e.g. the x86 architecture [Men05, RG05].

One challenge of the x86 architecture is that unprivileged instructions let the CPU access privileged states. Software running in the VM can read the code segment register to determine the processor's current privilege level. A virtualizable processor would trap this instruction, and the VMM could then patch what the software running in the VM sees to reflect the virtual machine's privilege level. However, the x86 does not trap this instruction. Hence, with direct execution the software would see the wrong privilege level in the code segment register [RG05].

A solution to enable virtualization on non-virtualizable architectures is provided by *para-virtualization*. Here, the VMM designer provides an "almost" identical abstraction, where non-virtualizable parts of the original machine instruction set is replaced by virtualized equivalents. The drawback is that any operating system run in a VM of a para-virtualized VMM must be ported to support the changed instruction set. However, most normal applications run unmodified. In contrast, para-virtualization provides better performance as the guest systems knows about the VMM and hence can be further optimized. An example for a VMM with para-virtualization is Xen.

Another possibility to overcome the virtualization problems of x86 architectures is direct execution with binary translation which is used by VMWare [AA06]. The advantage of binary translation is that any unmodified x86 OS can be executed in VMWare's virtual machines. Binary translation basically translates kernel code to

replace non-virtualizable instructions with new sequences of instructions which have the intended effect on the virtualized hardware.

Intel and AMD both developed hardware virtualization support for x86 CPU VMMs. With these technologies, full virtualization is possible on x86 CPUs, too. This means, Xen does not need to para-virtualize the architecture and VMWare can do virtualization without binary translation. However, these technologies often suffer lower performance, especially for I/O workloads and require further research [AA06].

2.1.2 Memory virtualization

The traditional technique to virtualize memory is a VMM which contains a data structure called *shadow page table*, a shadow of the virtual machine's memory management data structure. This enables the VMM to control which pages of the machine's memory can be accessed by the virtual machine. The VMM detects changes of this data structure and directs them to the actual page location on the hardware memory. Hence, a VMM can always control what memory each virtual machine is using. Furthermore, the VMM can page the memory of a virtual machine or parts of it to a disk. Therefore, the memory of a virtual machine can exceed the physical memory. The VMM can control how much memory each virtual machine shall get according to the needs and thus seems to be able to solve all memory problems, unfortunately with the drawback of performance loss.

One reason for such a performance loss is that the VMM operating at the hardware level cannot page parts of the virtual machines memory to a disk as effectively as it could be done by the guest operating system. This is because the guest OS is likely to have more and better information about the VMMs virtual memory system. Furthermore, running multiple virtual machines can waste memory because redundant copies of code and data (e.g. the OS or some applications) are stored across virtual machines.

Another disadvantage of shadowing arises in the field of high performance computing (HPC). There, a VMM with a small memory footprint is desired which is inhibited by potentially huge page tables. The memory footprint can have direct impact on modern execution platforms, e.g. multicore processors with shared cache [VNOS08]. One current research approach addressing these issue is the hardware support to store the VM's "memory map" directly on the hardware (e.g. Intel Extended Page Tables [EPT]).

In the future, better resource management can improve memory virtualization and hardware-managed shadow tables as presented in mainframe virtualization architectures could accelerate virtualization, too [RG05].

2.2 I/O virtualization in particular

A very basic but effective example of storage hardware (e.g. a normal hard disk drive, HDD) virtualization is provided by VMWare server, where the VM's HDD is either a simple file or partition inside the host OS. However, this device cannot be shared with other VMs. Hence, to realize sharing of I/O devices, e.g. the network interface card or storage devices, more and more challenges arise, especially when a performance efficient implementation is required.

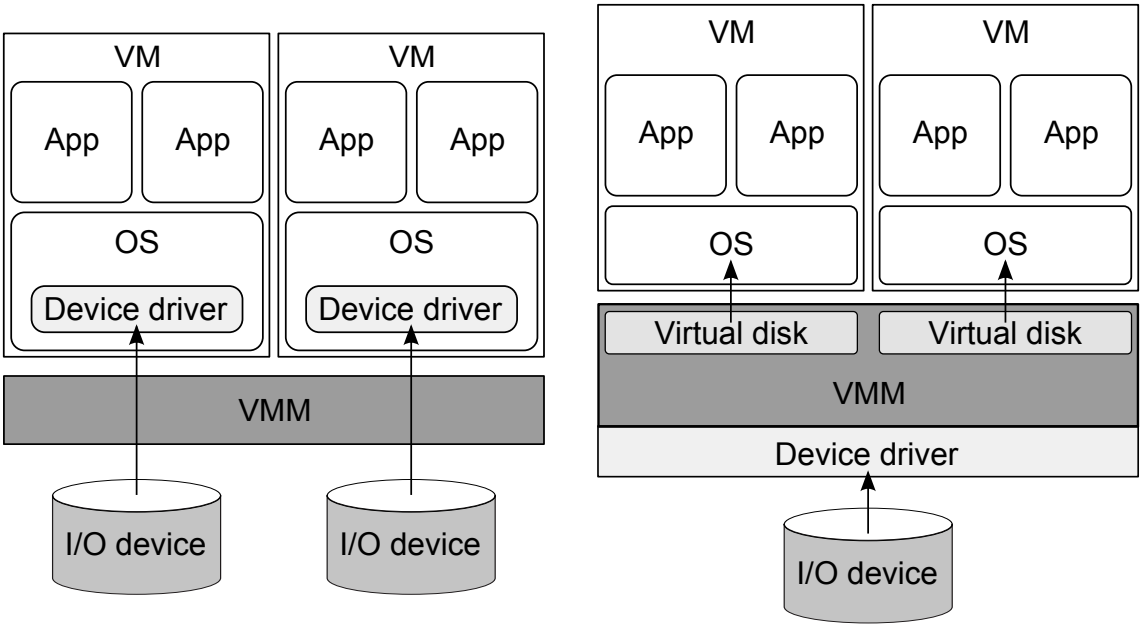


Figure 2.2: Pure isolation and shared devices

Basically, I/O virtualization means that the virtualization layer must offer a uniform interface to communicate with the I/O devices. Its advantage is that different VMs can use the device infrastructure simultaneously. Usually, all I/O devices are shared among all VMs of one machine (see figure 2.2) [KS08]. However, there are hypervisor implementations where each VM has its own devices - called *pure isolation* - for security reasons, e.g. the MILS separation kernel designed by the NSA [KS08].

In the case of pure isolation, performance is not relevant insofar as each VM has its own devices and does not share them. Hence, the performance is purchased by attaching additional devices. If one wants to save additional devices and build an I/O virtualization where devices are shared, performance efficiency comes back to the surface in every design but in a different way.

The classic I/O subsystems of IBM mainframes use a channel-based architecture. Access to I/O devices in this architecture is through communication with a separate channel processor. This reduces the CPU overhead inside the VMM. But even with I/O channels, the additional performance cost of I/O virtualization cannot be omitted.

In current technologies, it is the VMM which assures that all device accesses are legal and consistent and hence every I/O access must pass through the VMM. This can have the advantage that the VMM can serve as a multiplexer to control e.g. QoS. On the other hand, the VMM intervention increases I/O latency and CPU overhead and is the performance bottleneck in case of I/O intensive workloads. This overhead is extremely adverse in HPC, where interconnected I/O hardware provides very low latency and high bandwidth [LHAP06], especially the network interface cards.

To move this performance bottleneck out of the VMM, other approaches use a dedicated I/O VMM in a special privileged partition. An example is Xen. In this case, the dedicated I/O hypervisor itself often includes an entire OS. Although the work for I/O processing is now moved out of the actual VMM, this approach can decrease performance because the hypervisor must send the I/O requests to the dedicated I/O

VMM. Furthermore, there can also be additional costs to copy or map data to the I/O VMM. This again leads to the need of performance efficient I/O virtualization implementations.

The authors of [VNOS08] describe in their work the basic challenges and problems of implementing a system-level virtualization solution for HPC. One of the addressed issues is the previously motivated need for efficient I/O mechanisms and storage solutions for the virtualized environment, since overhead of current virtualization techniques can be unacceptably high. The proposed solutions and research for performance efficient I/O virtualization are versatile and will be briefly presented in the following. A detailed discussion of I/O virtualization performance analysis related to this work is given in chapter 3.

One approach uses self-virtualized I/O devices which are peripherals with additional computational resources close to a physical device. One of its responsibilities is to (de)multiplex a large number of *virtual devices*. To achieve this, the VMM intercepts all VM I/O requests to virtual devices and maps them to the corresponding physical device. This mapping is done by the additional computational resources and as such self-virtualized devices can be compared to the channels mentioned in the previous sections.

The *VMM-Bypass I/O virtualization* is based on the idea of para-virtualization and decouples I/O handling from the VMM. To virtualize a device in a VM, a special device driver called *guest module* is implemented in the guest OS. Since the guest module does not have direct access to the device hardware, the VMM implements a software component called *backend module* which provides device hardware access for the guest modules. The backend module can talk to the device directly or use the privileged module of the OS-bypass device driver which enables processes to directly execute I/O operations without the involvement of the OS kernel. This avoids overhead caused by context switches etc. [LHAP06]. Here, performance critical I/O operations can be carried out directly in the VMs without the need of the VMM and/or a privileged VM.

In contrast to centralized I/O virtualization - where the I/O virtualization is integrated into the VMM - there are other approaches using a separate I/O virtual machine (IOVM) dedicated to I/O virtualization only [WJW07]. The motivation is the same as for this work, namely to overcome the problems of centralized I/O virtualization. Some of these problems are that the VMM could be easily overloaded by I/O virtualization. A possible solution to these problems of centralized I/O virtualization is *scale-up* which means to add more resources to the centralized VMM. However, this does not necessarily result in increased I/O virtualization performance [WJW07], whereas *scale-out* (creating a separate IOVM) can provide a scalable solution (see section 3.1).

2.3 Outline of the System z I/O architecture

When talking about mainframes, one thinks of a special type of computer often used by large organizations. They are mainly used for business or research applications with high performance requirements like in the finance sector or weather forecast. A modern mainframe has special properties like its CPU power, I/O throughput, availability, reliability and robustness which predetermines it as a central server [Gre05].

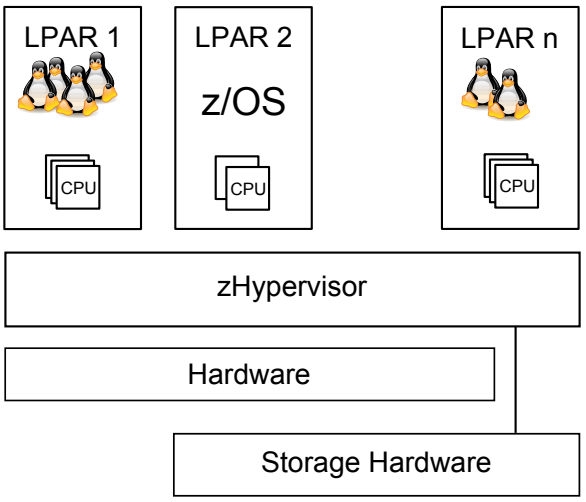


Figure 2.3: System z I/O architecture overview

Today, mainframes gain more importance because they have many benefits with respect to server consolidation like scalability, manageability and capacity flexibility.

The System z is a such a mainframe. Its hardware virtualization layer is the zHypervisor, depicted in figure 2.3. Basically, it virtualizes the mainframe's hardware into so called *logical partitions* (LPARs), i.e. each LPAR can be considered as a separate, smaller System z. In each partition, one can install an operating system (e.g. zOS). Furthermore, it is possible to run another hypervisor (e.g. z/VM) inside such a LPAR which can itself host several guest systems (in this work called clients). Each LPAR can be equipped with its own hardware configuration, including CPU power, memory size and storage devices. The storage hardware could be e.g. direct attached storage devices or a storage area network (SAN).

Currently, all I/O requests are sent to the hardware via the zHypervisor. Because the variety of storage hardware devices and vendors is permanently increasing, IBM discusses other possibilities of I/O handling. One possibility is the introduction of a dedicated virtualization layer for I/O, further explained in chapter 4.

2.4 Palladio Component Model

The Palladio Component Model[3] (PCM) is a domain-specific modeling language to specify component-based software architectures and their non-functional requirements in a parametric way. This model with its associated tool-set is developed by the SDQ-Group[4].The PCM approach originates from the concept of parameterized contracts introduced by R. H. Reussner [Reu01] with its idea to use *Service Effect Specifications* (SEFF) to specify the internal behavior of components. This approach developed and improved to a complex meta-model to describe software architectures with concepts like components, interfaces and connectors [BKR09].

The transformation of the component model to an ECORE instance [(EM09] realized by K. Krogmann [Kro06] enables model-driven tool support. There is a concrete graphical syntax first implemented by M. Uflacker [Ufl05] and improved by a student

[3]Palladio Component Model (PCM), http://www.palladio-approach.net/
[4]Chair for Software Design and Quality (SDQ), http://sdq.ipd.uni-karlsruhe.de/

group project [KB07]. Moreover, Becker integrated a model-based simulation environment for performance predictions [BKR09]. The advantage of such a simulation and analysis framework is that extra-functional properties of software architectures can be evaluated prior to the implementation. This can support in trading off design decision and thus prevent from costly design changes in late software development stages.

PCM defines an own, specific development process built on the component-based software engineering (CBSE) development roles [CD00]. This is possible because the Palladio component model is composed of a set of complete model parts according to the CBSE roles. An explanation of these roles and PCM is given in [BKR09] and more detailed in [Bec08]. The following sections introduces the main concepts of PCM focusing on the ones needed in this work.

2.4.1 PCM developer roles and their artifacts

In PCM, there exist four different developer roles. These roles are: the *Component Developer* who designs and models components, the *Software Architect* who assembles the components to a complete system, the *System Deployer* who allocates the components with hardware resources and the *Domain Expert* who models the user behavior. For each role PCM defines a specific view on a PCM model instance. This model instance is a combination of submodels, the views of each role. The design and implementation work of each role is stored in role-specific artifacts. Together, these artifacts form the complete PCM model. They will be explained in the following.

The Component Developer creates components by specifying their interfaces and their internal behavior. The artifact or part of the PCM instance resulting from this role is the *Repository*. It stores information about required and provided interfaces assigned to components and the *Resource Demanding Service Effect Specifications* (RD-SEFFs) which model the internal behavior of components [Bec08, Koz08]. Furthermore, components can be equipped with special parameters (*ComponentParameter*) influencing the behavior of the component. This special parameter type has the advantage that the Software Architect can override this parameter value in the later development process. In short, the *Repository* stores the information about components and their relationship to other components defined by interfaces.

The artifact of the Software Architect role is the *System Model*. It stores how Software Architectures are assembled of components specified in repositories. By connecting these components to a system, the Software Architect puts components into a context which is why such components are called *Assembly Context*. If necessary components are missing, he delegates their specification and implementation to the Component Developer. Moreover, the Software Architect chooses the best fitting component if the repository provides two components with identical interfaces but different behavior. Important to know is that he has no knowledge about the implementation of the components and looks at them as black boxes. His target is to assemble the complete software system.

Now, all components must be deployed on resources. The System Deployer specifies these *resource environments* by specifying their hardware resources like CPU(s), network latency etc. Furthermore, the system deployer allocates the components of the specified system onto the defined resources. The *Allocation Model* and the

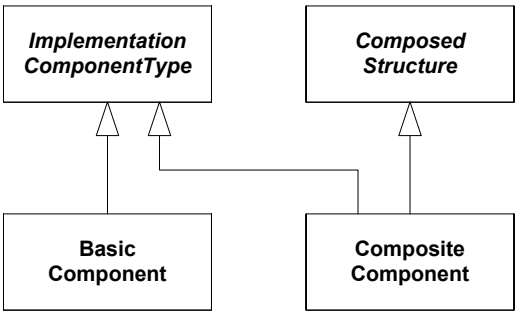


Figure 2.4: PCM component types

resource environments are artifacts of this development step. The resource environment contains the hardware resource information and the allocation model stores the information, to which resource the assembly contexts are allocated to.

The Domain Expert defines in special usage scenarios how the system is used, i.e. he provides information about the amount of users, their think time or the interarrival time of inquiries. These usage scenarios and their behavior are stored in the the *Usage Model* of the PCM instance.

2.4.2 Components, interfaces and datatypes

Interfaces can be used to describe which services a component offers or which services are needed to provide this service. Thus, they can be compared to the method signature of a programming language. Interfaces can serve as contracts between components, specifying the component's provided and required functionality and how they communicate with their environment. If a component offers a service, this interface is called *provided interface* and needed services are called *required interface*. These interface role types are explained in more detail in [RBK+07].

The most important component type in PCM is the *ImplementationComponentType*. This component type provides an abstract specification of the component's implementation. Such a component specifying all its required and provided interfaces could be a *BasicComponent* or a *CompositeComponent*. The behavior of a *BasicComponent* is set by its own behavior. A Component Developer uses this type, if the component cannot be decomposed further. In contrast, a *CompositeComponent* is implemented by composing other *BasicComponents* or *CompositeComponents*. Hence, the functionality of a *CompositeComponent* is specified by its inner components.

In PCM, signature lists of interfaces can contain different *data types* to express the data needed or returned. A data type can be one of *PrimitiveDatatype* which cover the common basic types of programming languages, a *CollectionDatatype* to represent collections like lists or sets or a *CompositeDatatype* to define a type which is a set of other elements. The latter can be used to define own data types.

2.4.3 Resource demanding SEFFs

The *Resource demanding service effect specification* (RD-SEFF) was introduced by Koziolek [Koz08] as a language to describe the behavior of component services. It was specifically designed for performance analysis. Such a language to describe each

single service instead of the complete architecture has the advantage that it exploits benefits of CBSE, such as re-usability and division of work.

Hence, an RD-SEFF can be used to describe the internal behavior of a component service (provided role) in an abstract manner. With RD-SEFFs, the Component Developer can specify dependencies to external services (required roles), can assign resource demands or he can pass variables to other components.

In principle, the RD-SEFF is a chain of different types of actions. This chain or control flow always starts with a *StartAction*, ends at the *StopAction* and can be structured by branches, forks or loops. Some of the other actions placeable between start and end will be explained in the following.

InternalActions represent any calculation or execution within a service that does not require external services. This calculation can be modeled by a hardware resource demand (e.g. CPU), a simple delay, a hard disk access etc. In any case the *InternalAction* uses a *ParametricResourceDemand* construct to issue this demand. The **ParametricResourceDemand** is used to specify which hardware is needed by this component, e.g. CPU, HDD, network. The Component Developer can only specify that the CPU is required, he cannot specify which CPU should be used. This separates the hardware model from the software model and enables the parameterization for different resource environments. The actual amount of the demand issued to a resource can be specified by a stochastic expression.

AquireAction and **ReleaseAction** can be used to acquire respectively release passive resources of a component. Components can have passive resources, e.g. to model semaphores, thread pools etc. The capacity of these passive resources can be parameterized as well.

ExternalCallAction defines a dependency between components by referring a required interface role of the current component and therefore corresponds to a synchronous service call. This action does not specify resource demands, but delegates them to the called service. Furthermore, it offers the possibility to pass values to the input parameters and receive values via the return type by using the *VariableUsage* construct.

SetVariableAction This action can be used to specify the returned value of a service or any other variable in the model. Currently PCM supports five characteristics: `VALUE`, `STRUCTURE`, `TYPE`, `BYTESIZE`, `NUMBER_OF_ELEMENTS`. With stochastic expressions it is possible, to assign values to these characteristics to better qualify the variables and parameters of the model.

Loops There are two different types of loops which are supported. The *LoopAction* models a repeated execution by specifying the iteration count by a stochastic expression. The *CollectionIteratorAction* models a repeated execution for each element of a *CollectionDatatype*.

Branches manipulate the control flow of the RD-SEFF. There are two types of branches to specify and alternative control flow. In a *ProbabilisticBranch*

the path of the control flow depends on a fixed probability whereas a *GuardedBranch* checks which of the specified conditions is true to determine which path the control flow takes.

Forks split the control flow in several separate control flows. Usually, the fork is asynchronous and the original control flow continues directly after all sub-RD-SEFFs are forked. However, *SynchronisationPoints* can be used to synchronize all forks. Then, the original control flow is blocked until all sub-RD-SEFFs are completed and then continues.

2.4.4 System

The *System Model* is the domain specific language of the software architect which can be used to assemble components of repositories to a fully functional software application [Bec08]. Like the *CompositeComponents*, the *System* is a *ComposedStructure* with inner components.

To the outside, the system offers *SystemProvidedRoles* and *SystemRequiredRoles*. The *SystemProvidedRole* can be used to interact with the system. This role can only be used by the *Usage Model* and not by other components and its call is delegated by a *DelegationConnector* to the matching interface of an inner component. The inner components of a system are connected by *AssemblyConnectors*. Each component of a system is put into an *AssemblyContext* by its specified connection and relation to other components.

Furthermore, the system keeps the information which component parameters specified by the Component Developer have been overridden by the Software Architect. These component parameters characterize the state of a component in an abstract and static way and hence offer a more flexible parameterization of the model.

2.4.5 Allocation

The *Allocation Model* specifies which components are allocated to which hardware. For this purpose the hardware environment must be specified in the *ResourceEnvironment* model and then the system can be deployed onto this *ResourceEnvironment*.

PCM supports the System Deployer by offering several hardware *ProcessingResourceTypes* (explained later in this section) to specify such a *ResourceEnvironment*. Basically, an hardware environment where the software system is executed can be specified by several *ResourceContainers*. The latter represents a server or any type of computer. These resource containers can be interconnected by *LinkingResources*, an abstraction of network connections.

As a *ResourceContainer* represents a computer, it provides one or more *ProcessingResourcesTypes* like CPU and HDD and *CommunicationResourcesTypes* like network interfaces. Each of these resources can be specified by *ProcessingResourceSpecifications*. This value defines the processing rate of a resource in abstract units, i.e. the System Deployer can define this unit (e.g. the amount of cycles per second of a CPU). The *LinkingResource* defines the properties of the network connections of the *ResourceContainer* which are throughput and latency.

By the mapping of the system's components to hardware resources the System Deployer creates the *AllocationContext* to component of the system.

2.4.6 Usage

The *Usage Model* serves as a description for different types of user interactions with the system. Each possible user interaction is captured in a *Usage Scenario*. As there can be several types of users with different types of interactions, a *Usage Model* can contain several *Usage Scenarios*.

The *Usage Scenario* specifies the *Usage Behavior* and the type of *Workload* that is issued to the system. The *Usage Behavior* defines a set of system calls and their order which is executed by the user or a group of users.

The *Workload* defines the frequency in which the *Usage Behavior* is executed. PCM distinguishes two types of workloads. In case of an *OpenWorkload*, the Domain Expert can only specify the interarrival time of new users. This is the time elapsing from the arrival of user one till the arrival of user two. After a user has executed its usage scenario, he leaves. In a *ClosedWorkload*, the Domain Expert can specify the amount of users executing usage scenarios and the think time a user waits before the next execution of his usage scenario.

2.4.7 Random variables, stochastic expressions and probability distribution functions

In PCM, most developer roles use *random variables* to specify performance properties, especially if one needs to characterize situations under uncertainty [Bec08]. Examples for such situations are the interarrival time, the amount of loop iterations or characteristics of input parameters. Sometimes one wants to use *mathematical expressions* and *random variables* to express new random variables. The Stochastic Expressions (StoEx) enables the model developer to use mathematical operators ($+, -, \cdot, /$ etc.) or logical operations (==,>,<,and,or, etc.) to calculate new random variables. The possibilities of stochastic expressions are explained in [RBK+07].

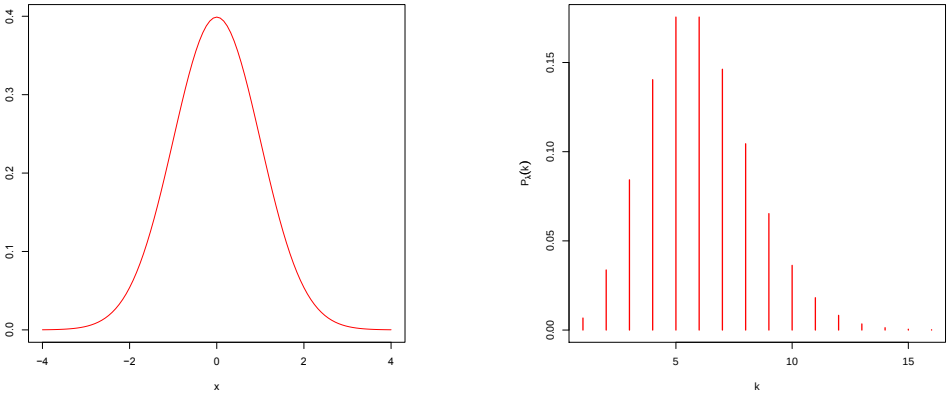
Not mentioned in the Technical Report [RBK+07] are the probability distribution functions supported by PCM. The following explains the functions which can be used within a StoEx. The functions supported are some of the most common probability distribution functions. Used in a StoEx, such a function returns a random variate according to the kind of the probability distribution and the specified parameter(s).

Defining own distribution functions

Already implemented in PCM and its StoEx editor is the possibility to specify probability mass distribution functions (PMF) and probability density functions (PDF). PMFs describe discrete and PDFs continuous distributions. An example for the syntax is `DoublePDF[(x;p)(y;q)(z;r)]`. A detailed explanation of the discretization of continuous distribution functions and the differences of discrete and continuous distributions is given in [RBK+07].

Uniform Distribution Function

PCM supports two types of uniform distribution functions. They can be distinguished by their return type. *UniInt(a,b)* returns a random variate of the `Integer` type whereas *UniDouble(a,b)* returns random variates of the type `Double`. Therefore



(a) Standard normal distribution, $\mu = 0, \sigma = 1$

(b) Poisson distribution, $\lambda = 6$

Figure 2.5: Probability density function of the standard normal distribution (a) and the Poisson distribution function (b)

UniInt(a,b) is a discrete probability distribution and *UniDouble(a,b)* a continuous probability distribution.

In case of *UniInt(a,b)* the returned variate is an integer value out of the interval $[a, b]$. For example, $UniInt(1, 4)$ returns $1, 2, 3$ or 4, each value with a probability of 25%.

The expression *UniDouble(a,b)* returns a uniform random variate out of the interval (a, b), where a and b are real numbers with $a < b$. The density function of *UniDouble(a,b)* is

$$f(x) = \frac{1}{(b-a)}, \quad a \leq x \leq c$$

Normal Distribution Function

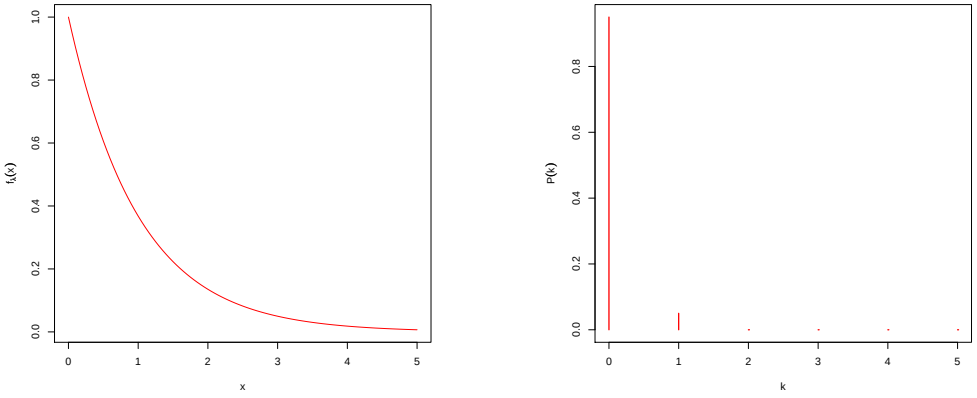
If the expression $Norm(\mu, \sigma)$ is used, this function returns a normally distributed random variate. The continuous probability distribution is determined by two parameters, mean (μ) and variance (standard deviation squared, σ^2), where $\sigma > 0$. The density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R}$$

Figure 2.5a gives an example for the probability density function of the standard normal distribution ($\mu = 0$, $\sigma = 1$).

Poisson Distribution Function

The expression $Pois(\lambda)$ returns a random variate of a discrete probability distribution, called Poisson distribution. It expresses the probability of a number of events



(a) Exponential distribution probability density function, $\lambda = 1$

(b) Binomial distribution probability mass function, $n = 1$ and $p = 0.05$

Figure 2.6: Probability density function of an exponential distribution (a) and probability mass function of a binomial distribution (b)

occurring in a fixed period of time if these events occur with a known average rate and independently of the time since the last event. This distribution function is characterized by its parameter λ, where $\lambda > 0$. This parameter is equal to the expected number of occurrences during a given interval. If the expected number of occurrences in this interval is λ, then the probability that there are exactly $k \in \mathbb{N} \cup \{0\}$ occurrences is equal to

$$P_\lambda(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

An example for a Poisson distribution with $\lambda = 6$ is given in figure 2.5b.

Exponential Distribution Function

The stochastic expression for an exponential distribution function in PCM is $Exp(\lambda)$. It is a continuous distribution function with the probability density function of

$$f_\lambda(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Here $\lambda > 0$ is the parameter of the distribution, often called the rate parameter. An example of a probability density function of $Exp(1)$ is given in figure 2.6a.

Binomial Distribution Function

The binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p. The PCM expression to generate a random variate of a binomial distribution function is $Binom(n, p)$. The probability of getting exactly $k = 0, 1, 2, \ldots, n$ successes in n trials is given by the probability mass function

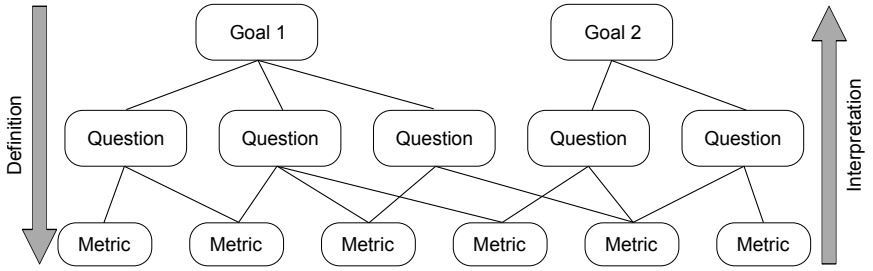


Figure 2.7: Relation between goals, questions and metrics [Hap08].

$$P(k) = \binom{n}{k} p^k (1-p)^n$$

Figure 2.6b shows the probability mass function of a binomial distribution where the number of experiments $n = 1$ and the probability of success $p = 0.05$. This setting can be compared to a an unfair coin toss, where the probability of observing heads (1) is only 5%.

2.5 The Goal/Question/Metric approach

The following describes some of the crucial facts of the Goal/Question/Metric approach (GQM) introduced by Basili, Caldiera, and Rombach [BCR94]. This approach guides the method of Happe to derive software performance models [Hap08], explained in section 2.6.

GQM is a systematic top-down process model to quantify the quality of processes or products. The major purpose of this approach is to attain a greater target and to improve the understanding of this target. By specifying a *goal*, refining it by *questions* and defining *metrics*, one documents step-by-step the motivation, the approach and the technique of a measurement to quantivy the greater goal. Basili, Caldiera, and Rombach argue that measurements must be goal-oriented and defined in a top-down manner to be meaningful and efficient. For example, if concrete goals are missing, the risk of collecting large amounts of unnecessary data is likely.

In GQM, *goals* strongly depend on the context where measurements shall take place. This defines the object, reasons, points of view, and environment of the measurements. Possible objects of measurements are products, processes, or resources. GQM requires the explicit definition of the goal's issue, object or process, viewpoint, and purpose to place a goal into a given context. *Questions* determine the assessment of a specific goal. They characterize the object of measurement (product, process, resource) with respect to selected quality attributes from the selected viewpoint. *Metrics* assign a set of data to each question to answer the questions in a quantitative way. GQM distinguishes objective and subjective metrics. The first depend only on the object under measurement (e.g. lines of code), the latter depend on the viewpoint from which the measurements are taken (e.g. readability of a text).

The GQM process starts with the definition of one or more explicit measurement goals. Questions refine the goals and identify its major components the measurements shall answer. The questions are further refined by metrics (see figure 2.7).

After the measurements are taken, the collected data is interpreted bottom up. Metrics are directed towards a specific question and help to interpret the measurements and to answer questions. Whether a goal has been attained or not can be decided by analyzing and interpreting the data with respect to the questions refining a specific goal.

2.6 Experiment-based derivation of software performance-models

Creating accurate performance models of complex software systems is often not easy and erroneous. Therefore, a systematic approach to i) identify performance relevant features of the test system, ii) design accurate performance models of these features and to iii) validate the prediction accuracy of the created model is necessary. Happe describes such an approach, called experiment-based derivation of software performance-models [Hap08]. This approach is inspired by general ideas by Jain [Jai91] and combines existing knowledge about the test system with iterative, goal-oriented measurements.

The motivation of such an approach is to avoid several common mistakes in software performance evaluation. The most common mistake is the absence of goals. Furthermore, approaches are often unsystematic and lead to unnecessary high effort and inaccurate performance models. The choice of the modeled factors must be problem-driven and not by the analysts knowledge.

Moreover, performance models must be validated. This can be accomplished by comparing the results with expert intuition, theoretical results or real system measurements. The proposed method uses the latter, as expert intuition can be misleading and theoretical results can be erroneous, too. To verify the model, performance analysts must identify necessary assumptions about the test system and validate them. This allows to focus on the most influential factors [Hap08]. The following explains the experiment-based derivation of software performance models in more detail and explains, how assumptions and their evaluation lead to a validated performance model.

The method

The design of accurate and reliable performance models require a systematic model design based on validated assumptions [Hap08]. This approach can be used either by performance analysts to design configurable models or by software architects to create prediction models for existing parts of the system.

Performance model design is driven by a specific goal. This helps to focus the design effort on the most relevant factors and keeps the model on an abstract level. The goal is defined by *purpose, issue, object* and *viewpoint*, similar to the Goal-Question-Metric (GQM) approach [BCR94].

The *purpose* sets the general goal of this method, e.g. designing a configurable performance model. The goal can be focused on special characteristics of the test system by specifying *issues* like different configurations. *Objects* target the effort towards a specific part of the system, e.g. the requests handling. The perspective for

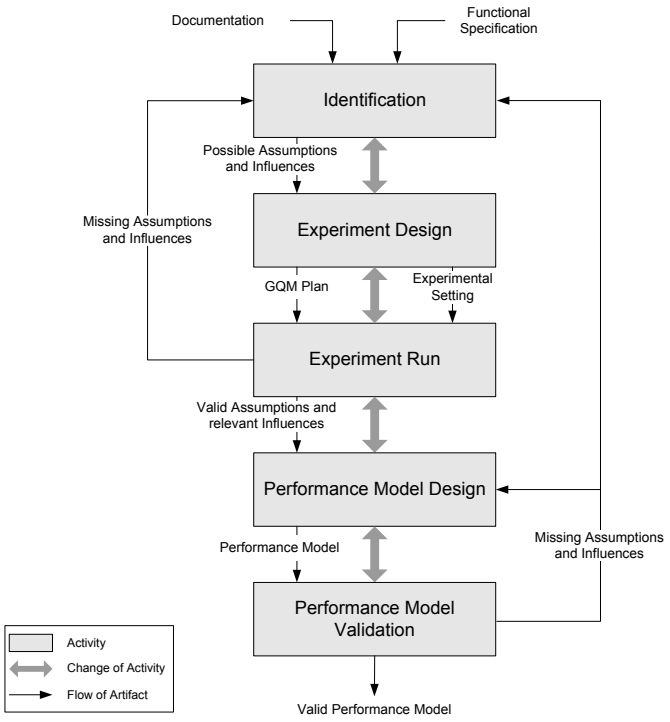


Figure 2.8: The different steps of experiment based derivation of software performance models [Hap08].

which the performance model shall be made is specified by the *viewtype*, for example a special user group or another part of the system.

Figure 2.8 depicts the process model of this method. All steps are executed iteratively and refine the performance model by adding additional assumptions or performance-relevant factors.

Identifying performance relevant factors

In its first step, the proposed method aims to identify an initial set of performance relevant factors of the test system. According to the QGM schema, performance analysts pose questions to address these factors. These questions are based on documentations specifications. For example, the size of a request issued to the VL may influence its performance, and hence a performance analyst may ask: "How does the request size influence the throughput of the VL?".

As documentation and specification focuses on the description of functional features of the test system, it may be difficult to judge whether a specific factor influences the performance or not. Hence, this first step lists all performance relevant factors of interest for the modeling goal. The next step systematically identifies those with influence on the system's performance.

Experiment design

The experiments designed in this step aim for the systematic evaluation of performance influences of the factors collected in the previous step. The GQM method therefore defines questions and performance metrics. Its extension by Happe adds

specific scenarios and hypotheses to evaluate the experiment results ([Hap08], section 4.1.3).

Scenarios define the experimental setup. In software performance evaluation, they have to be defined explicitly and must be representative to evaluate the influence of specific factors. As questions of a GQM plan often leave several degrees of freedom, the scenarios introduced by Happe fix the experimental environment (e.g. execution environment and other settings) which enables the experiment to be reproducible. In contrast, carelessly chosen scenarios can lead to wrong conclusions from the measurements, as the performance influences of specific factors ar likely to depend on the experimental setup.

Hypotheses formulate the expected outcome of the experiment for each question and are based on available documentation of the test system. Hypotheses support the performance analyst to answer the questions posed in the previous step. Therefore, hypotheses should be revisable and hence they should be formulated in such manner that measured results allow conclusions, whether a hypothesis needs to be revised or not.

In short, scenarios allow the performance analyst to define specific and reproducible experiment setups. Hypotheses formulate the expected outcome of an experiment and used to assess whether a performance factor conforms to the expectations or not.

Experiments

Now the previously defined experiments can be executed to measure their results. If these results conform to the expectations made, the assumptions made can be considered as valid, until proven otherwise. If he measurements differ from the expected results, one needs to examine the cause and more detailed evaluations might be required.

After all assumptions made in the previous step are valid, the performance analyst can use this validated assumptions about the influence of performance relevant factors to create a performance model of the test system. Then, the performance model can already be considered as performance valid.

Performance model design

After the previous step is completed, the performance analyst is able to design a performance model. He can use the results of the previous step to decide which factors are performance relevant and shall be integrated into the model. Moreover, experiment results can be used to quantify the resource demands depending on the experiment settings.

At this stage, the performance model reflects the performance relevant factors of the test system in an abstract manner [Hap08]. But if the model represents a valid abstraction of the performance behavior of the complete system or not must be evaluated in a separate validation step.

Model validation

As the creation of performance models for complex software systems is accompanied by several risks, this separate model validation step ensures that the model predicts the necessary performance metrics with the expected accuracy and that all performance relevant factors are modeled correctly. Some of the most important risks are that not all performance relevant factors have been identified or that their influence on each other was considered to be independent but were not. Moreover, a model validation can identify modeling errors which can be easily introduced in complex performance models. If the outcome of this step shows an unacceptable inaccuracy of the performance model, the performance analyst is required to refine or adjust the model. This can even require the definition and execution of further experiments.

The model validation itself employs the same scenario-based GQM method as the experiments design. But in this case, the hypotheses make statements about the expected prediction accuracy of the model. Although it might be intuitive to keep the prediction error of the model as low as possible, it is sometimes acceptable to allow a certain inaccuracy to keep the performance model simple but still achieves a moderate prediction accuracy.

The performance model validated reflects only factors identified in the previous steps and hence the validation cannot make statements of the validity of other scenarios or factors. Therefore, to allow a generalization of the prediction model while ensuring the prediction accuracy, the experiments should reflect a wide range of different scenarios and environments.

2.7 Summary

This chapter gave an overview about virtualization in general and the idea of how I/O virtualization in case of an IBM system can be accomplished. It discussed the current virtualization technologies and depicted how they are related to the I/O virtualization for IBM systems. Furthermore, it presented PCM as a tool for performance analysis of such I/O virtualization architectures. Moreover, a methodology to systematically create performance models was proposed.

3. Related Work

This chapter introduces other research projects dealing with performance analysis, performance modeling and I/O virtualization. The first section presents current research in the field of I/O virtualization and what kind of performance analysis currently exists. The next section presents performance modeling projects using PCM as the preferred performance model. Although not directly related to I/O virtualization, the last section proposes alternative performance modeling approaches based on queuing networks.

3.1 I/O virtualization performance analysis

There are several research projects dealing with I/O virtualization and its performance. They are not directly related to this work as they analyze the performance of I/O virtualization on a different level of abstraction. Nevertheless, they provide useful ideas and insights on how to implement I/O virtualization. Basically, both proposed approaches try to improve the I/O virtualization's performance by either by scale-up (e.g. additional hardware) or scale-out (moving the performance bottleneck to dedicated hardware). The following introduces these approaches and discusses their advantages and drawbacks.

As scale-up, the Wei et al. propose a solution for delivering scalable network performance on a multi-core platform [WJW07]. They focus on the so-called *driver domain* which acts as a proxy between the guest OS inside a VM and the physical device. The most common way to deploy such a driver domain is to use the hypervisor as a centralized driver domain. Now, the authors want to achieve a performance increase by moving the I/O virtualization work out of the hypervisor onto dedicated I/O virtual machines (IOVMs). Their experiment results and performance comparisons show improved efficiency and flexibility of dedicated IOVMs compared to a centralized solution. However, the results must be restrained as the focus of the authors and their evaluation is on network virtualization on multi-core platforms and Xen as VMM.

The IOVM proposed by the authors is a specialized guest operating system which contains native device drivers for the physical devices and backend drivers for the

guests. Furthermore, they propose and compare three different configurations of such IOVMs: i) monolithic IOVMs, where all devices are assigned to a single IOVM, ii) multiple small IOVMs, where each device is assigned to one dedicated IOVM and iii) a hybrid configuration with multiple IOVMs where each manages a subset of physical devices. Furthermore, they explain how such an IOVM can be minimized (kernel size, network protocol stack, runtime).

Their performance analysis results provide new insights on the influence of different IOVM configurations, showing that IOVMs result in better performance (throughput and efficiency) compared to centralized I/O virtualization architectures. Another interesting detail of the experiment analysis is that the scheduling technique used to balance the workload across multiple cores can have an impact on the performance metrics. Especially if the IOVM CPU is the performance bottleneck, using a credit-based scheduler resulted in better efficiency than a static scheduler. The influence of scheduling strategies is of interest for this work, too, especially on multi-core platforms. However, the performance analysis of the proposed work remains on a more abstract level, namely the comparison of different IOVM configurations, whereas this work deals with the performance modeling and analysis of IOVM's internal configuration, like threads and communication model.

In another paper by two of the authors of the previous approach, they follow the approach of scalable networking from another point of view [WRJ07]. This time, the authors present a *scale-up* strategy for network I/O, based on the IOVM idea also used in the previous section. Furthermore, they present empirical results and analysis of such an implementation. In contrast to the previous work, this approach focuses on the IOVM internal setup and configuration, particularly the threading model.

Usually, the network backend driver in a centralized approach runs in the IOVM and is simply a Linux kernel driver, providing a way for network devices to be shared. It has an interrupt handler and also two contexts (receive and transmit) for special kernel tasks to run in. The transmit context is scheduled when a guest is transmitting data and is also responsible for handling completed transmits. The receive context processes data received from the NIC and moves them to the destination guest. This data move requires a specific amount of processor cycles and it is easy to saturate a single processor with a relatively low number of guests. One starting point for performance increase on multi-core platforms is possible to split the contexts across different cores. Furthermore, the authors vary the type of threads executed in the contexts to examine their influence. They distinguish two types. The first is the tasklet which is scheduled from an interrupt handler and must run on the same CPU that serviced the interrupt. Workqueues on the other hand are full kernel threads with an associated state.

In several experiments the authors examine different configurations of cores, context distributions and thread types. Results show that splitting the transmit and receive operations to separate cores provides no gain. Moving from a single processor to a SMP configuration further improves throughput, However, the overall system efficiency is decreasing. Adding cores to the IOVM showed that the number of L2 cache misses rose significantly. The performance analysis conducted in this work is closer related to this project as the previous one, as it examines the impact of thread types (and threading models) on a multi-core platform. However, it does not create

a performance model. Nevertheless it provides some interesting facts which can have influence on the architecture of IOVMs in general and the VL in particular. The main focus of this thesis is to create a performance model which allows to modify the model and adjust it to such architectural decisions to predict their performance impact.

Although the two approaches reside on a different level of abstraction, the scale-out approach demonstrates and confirms that the introduction of a separate IOVM is a promising approach, whereas scale-up provides moderate improvement, only. The fact that the scheduling technique and the thread types used can have a performance impact should lead to further considerations about the VL architecture proposed in this work. Furthermore, it legitimates the comparison of different threading models this work wants to draw. However, these research projects cannot contribute directly to this work by supporting the performance model creation.

3.2 PCM and performance modeling

There are two other research projects which elaborated a performance model of existing systems using PCM. The following sections describe these projects and the advantages and shortcomings of PCM emerged during these work.

3.2.1 PCM and CoCoME

The Common Component Modelling Example[1] (CoCoME) is targeted to provide a common component-based system as a modeling example to evaluate and compare the applicability of existing component models (like DisCComp, Fractal, Focus or UML Extensions). However, many of these component models concentrate on different yet related aspects of component modeling. Although CoCoME is an imaginary example, it describes use cases as they were delivered by a business company and could be reality. Moreover, there exist implementations [HKW+07] of this theoretical example.

In [KR08], the authors present the modeling of CoCoME with PCM. The intention of this project is to evaluate the applicability and capability of the Palladio approach with the CoCoME. Hence, the paper describes the advantages as well as limitations and shortcomings of PCM. The results are possible performance predictions of the non-functional properties of the CoCoME and further necessary enhancements or features for PCM to improve its practicability.

As PCM is primarily targeted at business information systems, the authors focused on modeling components, interfaces, resource environment etc. with respect to non-functional properties. With PCM's big strength in *sizing scenarios* (estimating performance under changing conditions, e.g. the usage scenario) and *relocation scenarios* (reuse of existing components in a different context) the authors were able to use their simulation approach to evaluate the response time of CoCoME's PCM model instance. For example, the authors showed the upper limit of concurrent service requests the system can handle based on the specification of extra-functional properties stated by CoCoME [KR08]. Furthermore, their work demonstrates how PCM supports the prediction of QoS properties, namely performance.

[1] CoCoME, http://agrausch.informatik.uni-kl.de/CoCoME

However, modeling the PCM instance of CoCoME exposed shortcomings of PCM, even when modeling an architecture of its target domain, the business information systems. Some of these limitations are important for this work, too, and will be discussed in detail in section 5.1. For example, the fact that components are stateless or no support of dynamic architectures complicated the modeling process and effect the modeling of this project, too. Hence, this work can profit from the limitations and solutions presented in [KR08]. However, the CoCoME PCM instance can be differentiated from this work for one main reason, namely it does not model a real-world scenario. This means, the results found in their work and the implementation itself where never used in the field. The work proposed in the following addresses this problem by evaluating the utilizability of PCM in an industrial environment.

3.2.2 PCM in industrial context

In his diploma thesis, Roman Andrej uses PCM to model an appropriate software system of the CAS Software AG[2] [And08]. The goal of this work was to study and evaluate PCM in a real world scenario. Therefore, performance predictions were made with PCM and compared to measured values of a real system to demonstrate the quality of the prediction. Furthermore, this work served to identify shortcomings of PCM when used in the field.

In his work, the author describes the construction of a performance model of the CAS teamCRM software system. He uses the GQM approach explained in section 2.5 to get an overview of the performance behavior and to determine the performance factors of the software system. The evaluation then describes the comparison of the model's performance prediction with the real system. The results demonstrated that PCM is a mature method for performance analysis in the domain of business information systems.

Moreover, the proposed work demonstrates the applicability of PCM and lists its advantages and shortcomings. These identified limitations and their possible solutions are important for this work, too. Although PCM's applicability and quality of performance analysis is shown by the proposed work, it is different as the requirements of this project are in a completely different domain, totally apart from business information systems. Hence, this approach can present new findings and insights in the applicability of PCM in "PCM-foreign" domains.

3.2.3 PCM and concurrent, message-oriented communication

The VL basically introduced in section 2.3 and described in detail in section 4 can be considered as a sort of message-oriented middleware, as its responsibility is the delivery of messages, sent by clients to the receiver, namely the storage hardware. Furthermore, its architecture and hardware predestines the VL to handle messages concurrently which is necessary to guarantee a most effective way of message handling.

The diploma thesis of Friedrich [Fri07] deals with such concurrent, component-based software systems, in particular the Java Messaging System (JMS). The author evaluates design patterns for concurrent systems with respect to their applicability for PCM. Hence, these results can be of interest for the VL architecture design.

[2]CAS Software AG, http://www.cas.de/

Both Friedrich and Happe [Fri07, Hap08] extensively evaluate the behavior of the design patterns, especially their performance relevant parameters. Furthermore, Happe proposes a systematic approach explained in section 2.6 to evaluate performance relevant factors. This approach will be applied in this work, too.

3.3 Performance modeling using (layered) queuing networks

As presented in the previous sections, there are projects analyzing I/O virtualization performance on a higher level of abstraction and hence with a different approach. During this work there were no research projects found which are related to this work on the same level of detail, analyzing or modeling the performance of an I/O virtualization approach. However, there are several research projects either modeling comparable or transferable software architectures. Some of them will be presented in the following.

Harkema et al. present a quantitative performance model of CORBA-based middleware [HGvdMH04]. Their goal is to apply their model to predict the performance of middleware-based applications. Although middleware is not related to the topic of this work, they have a lot of issues in common. For example, the authors have to deal with thread pool sizes, threading models and resource sharing, too. Furthermore, it encompasses a similar set of factors like processor speed or arrival rate. The authors describe the creation of a performance model based on queuing networks. This model is validated by comparing simulation results with performance results of lab experiments. They show that the performance prediction results match accurately and conclude that model-based approach to predict middleware performance is promising.

The paper of Ramesh and Perros describes a multi-layer client-server queuing network model with synchronous and asynchronous messages [RP00]. Their work is motivated by CORBA, too, where distributed objects use the client-server interface to communicate by synchronous and asynchronous messages. In particular, the model considers blocked clients, too. 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. The good accuracy results show that this novel queuing network can open up a new way to analyze queuing networks with blocking. Hence, their work indicates that other models can provide a way to analyze the performance of I/O virtualization, too.

The Proactor/Reactor patterns used in middleware [SHB07] are closely related to the synchronous and asynchronous I/O request handling analyzed by this work. Especially the Proactor pattern effectively encapsulates the asynchronous mechanisms of the operating system. Praphamontripong et al. present a qeueing model which captures the performance relevant characteristics of this Proactor pattern [PGGG06]. They analyze the performance of a Proactor-based Web server, concentrating on metrics like throughput and response time. The authors demonstrate the application of their model in practical scenarios and illustrate the use of the model to guide e.g. design-time configuration decisions. In a second paper, the authors

present a performance model of the Reactor pattern based on Stochastic Reward Nets (SRN) [PGGG07]. Their analyses are the same as in the previous paper and evaluates the same performance metrics as response time, throughput and loss probability. Again, a case study illustrates the use of the model. However, their performance models for both patterns are different and they only focus on either the Proactor or Reactor pattern. So far, no analytical comparison of both patterns was made which is one of the targets of this work.

The here presented works show good results and indicate promising applicability. Hence, the approaches of (layered) queuing networks models seem appropriate to model the rather hardware-oriented requirements of this work. However, the great benefit of the component-oriented PCM is its flexibility. It enables easy exchange components (e.g. synchronous with asynchronous implemented components) to observe the model behavior. Though appearing more complex, PCM seems also more comprehensive as it allows easy deployment and deployment changes, components or hardware configuration changes. Furthermore, it provides detailed graphical depiction of resource (e.g. CPU usage) or response times of components.

3.4 Summary

There are several research projects dealing with the performance of I/O virtualization. However, they highlight I/O virtualization from a different point of view then this work. Their focus is on the analysis of new I/O virtualization approaches in comparison to existing techniques. Unfortunately, there are no projects dealing with performance modeling of I/O virtualization. Other research projects like QN are promising approaches to create performance models, too, but were demonstrated in other domains and did not analyze I/O virtualization.

4. Virtualization layer architecture

This chapter gives a detailed explanation of a potential Virtualization Layer for I/O (VL) architecture which could be integrated into an IBM system. The following explains involved components, environmental requirements and restrictions, and design alternatives. First, an overview of a potential VL integration is given. The second section describes the environment, especially the hardware the VL is equipped with. Furthermore, it depicts all components the VL interacts with. The next section explains the internal architecture of the VL, distinguishing between two potential implementation alternatives, namely synchronous and asynchronous I/O. The last section gives an overview of the parameters determined to be relevant for this performance modeling approach.

4.1 Execution environment

One possibility to integrate a VL into an IBM system is as an application in an operating system within a privileged partition (see figure 4.1). Its target is to handle I/O requests and to send them to the storage hardware. As other partitions, the VL partition can be individually equipped with CPUs. It is privileged because it requires memory access to the other LPARs which facilitates request handling. In the remainder, VL is used as the expression for this partition, including both the OS and the request handling application unless stated otherwise. The term I/O interface encompasses all services offered by the OS to enable request handling.

From the user's point of view, the VL offers virtual devices the clients can be configured with. Clients are the guest systems hosted by LPARs (see section 2.3). Internally, the VL directs the requests to the attached logical device provided by the storage hardware. In turn, logical devices are related to the actual physical devices of the storage hardware.

The VL is a separate privileged partition. These privileges are crucial because the VL must have read and write access to dedicated memory areas of clients. These privileged accesses allow the VL to simply reset pointers to memory areas which avoids store and forward of requests. For example, to send a request to the I/O interface, the VL passes the pointer to the specific memory area of the request to

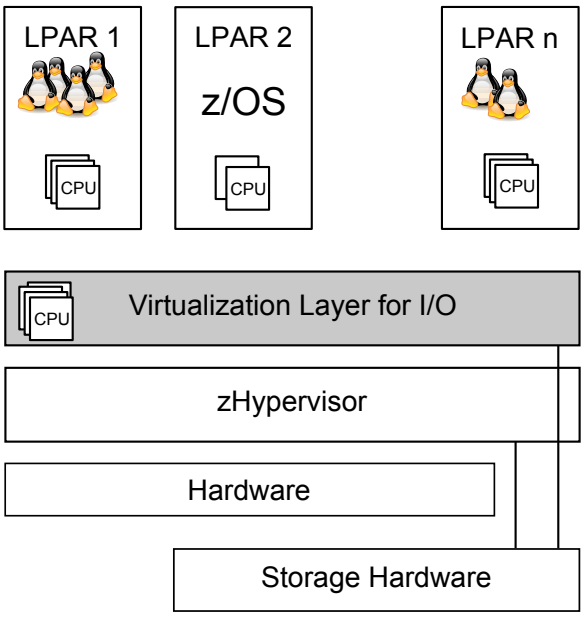


Figure 4.1: Architecture overview

the interface. This saves CPU time because requests do not have to be copied from the client memory to the VL memory. Thus, the processing time of the I/O thread to send an I/O request to the I/O interface is independent of the request size.

An advantage of a separate partition for the VL is that it can be equipped with an arbitrary amount of CPUs of the system's hardware. The CPU amount must be set prior to the execution of the VL. If CPU or CPU demand is mentioned in the remainder, this always refers to the CPU assigned to the VL if not mentioned elsewise.

Basically, the VL provides access to the storage hardware for a variable set of clients (see figure 4.2). Every client is configured with a different, varying amount of virtual devices served by the VL. The configuration of clients with virtual devices may change during the VL's execution as well as the amount of clients. Assigned to each virtual device is the *Request Completion Queue Pair*. This is a dedicated part of the client's memory the VL is allowed to access because of its privileges. Such a Request Completion Queue Pair is used to signal new I/O requests and receive their corresponding completion signal. The access of this Request Completion Queue Pairs and the processing of the requests is specified by the implementation of the VL, explained in the next section.

If a request is passed to and handled by the I/O interface, the OS providing the I/O interface uses the same CPU as the other components of the VL. Another property is its throughput which is restricted to a special amount of requests per unit of time. If the arrival rate is higher than the throughput rate, all further requests will be delayed until capacity is available. For example, if the throughput is x Requests per second and $x + 1$ requests arrive in second one, then the request $x + 1$ will be delayed. This throughput capacity is defined by the channel attached to the I/O interface. If necessary, it is possible to add more channels to the I/O interface.

Below the VL (see figure 4.2) is the storage hardware which can be accessed via the operating system's I/O interface. The storage hardware receives the requests sent

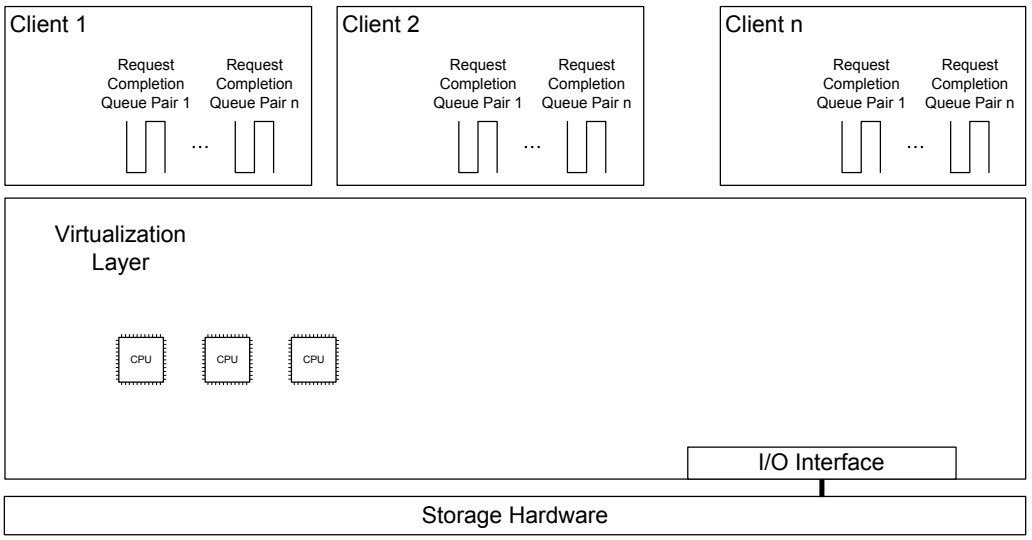


Figure 4.2: Execution environment of the VL

by the VL via the I/O interface and executes them. When a request completes, the results are signaled via the I/O interface. In this approach, the storage hardware acts like a black box, where only response times are of interest.

As I/O virtualization is performance critical, it is obvious to research for the best implementation and to determine the benefits and drawbacks of different design alternatives. Important and interesting questions are, what are the factors influencing the performance of such an implementation, how are they influencing the performance and whether another design (e.g. synchronous vs. asynchronous request handling) yields better performance results.

4.2 Virtualization layer internals

The VL is responsible for collecting and processing the requests of all the different client request queues. It preprocesses the requests, so they can be handled by the storage hardware. After a request completes, the results are sent to the respective client's completion queue. This can be achieved in mainly two ways. The first is based on synchronous I/O request handling, the second approach uses asynchronous request handling. The peculiarities of these two approaches will be explained and discussed in the following.

4.2.1 Synchronous request handling

This section explains the synchronous request handling process by two different point of views. The first is a static view, focusing on the involved components and the architecture. The second is a dynamic view, explaining the behavior of the different components.

4.2.1.1 Static view

In the synchronous case, each client has a request queue for each configured device which the client can fill with requests the device has to execute. An I/O thread

collects this request and sends it to the storage hardware by using the VL's I/O interface. Then, the I/O thread waits for the result of the request. When this arrives, he will return this result to the client. Hence, the client receives his response directly from the I/O thread and can now continue his work.

The specific amount of I/O threads must be large enough to handle all requests in time without blocking or delays. For this work, the number of synchronous I/O threads running is assumed to be infinite and independent of the amount of attached clients.

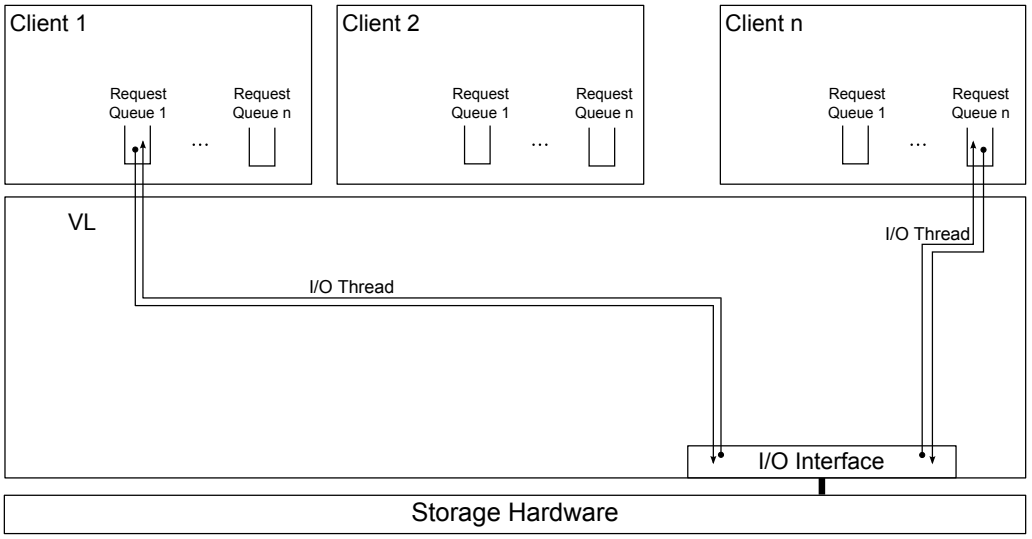


Figure 4.3: Architecture of the VL using synchronous request handling

4.2.1.2 Dynamic view

Whenever a client wants to send a request to the storage hardware, an I/O thread collects this request from the request queue and sends it to the I/O interface. Then, the I/O thread has to wait for the results of the storage hardware. This implies that the thread and hence the request issuing application are blocked until the result is returned by the storage hardware. This is the substantial difference to the asynchronous request handling explained in the following section. This delay from the arrival of a request until the storage hardware signals the completion is composed by two parts. The first part of the delay results from the request processing inside the operating system the I/O interface runs in. Th processing can be influenced and can take much longer if the CPU is used by other components and must be shared. The second part is the delay which is needed to actually read/write the data from/to the storage hardware.

As the VL and hence its I/O threads has privileged access to the client's request queues, the I/O thread can simply pass the pointer of the request's memory area to the I/O interface. This demands only a constant amount of CPU time. This constant time can only be influenced by other components, competing for the same CPU resource.

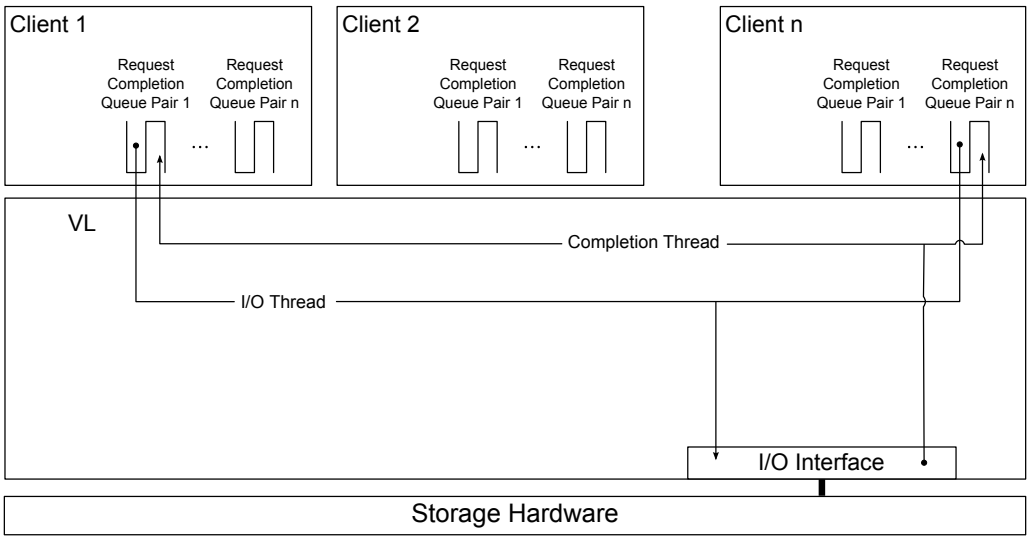


Figure 4.4: Asynchronous request handling: I/O threads and completion threads

4.2.2 Asynchronous request handling

Again, this section and the following section is composed of two different point of views, likewise the previous section. The first section discusses the static architecture, the second the dynamic behavior.

4.2.2.1 Static view

Again, one core element of the VL architecture is the I/O thread, depicted in figure 4.4. As in the synchronous case, I/O threads collect requests and pass them to the I/O interface. To issue requests and to receive completion signals, each device of each client uses the request completion queue pair.

The second elemental part of the VL are the completion threads, run by the VL whenever a request completes. They signal the completion of the request by putting the signal into the completion queue of the request completion queue pair the request was sent from. The completion threads are started by the VL, whenever a request completes. The I/O interface is more like a service component, offering to read from and store data on the storage hardware.

4.2.2.2 Dynamic view

A more detailed picture of the I/O thread's behavior is given in figure 4.5 and figure 4.6. The activity diagram (see figure 4.5) shows that the I/O thread, once started, runs until it receives an interrupt signal, e.g. because the VL will be shut down. While the I/O thread is running, it iterates in an infinite loop over all request queues the clients are configured with. It tries to access each queue to collect all requests currently in this queue. While one I/O thread is working on a queue, no other thread from within the VL can access this queue. Therefore, each further access of an I/O thread results in a blocked access and a delay, until the queue is released. Queue accesses from client's side are not blocked. This can be realized by a special design of the queue (see figure 4.7).

After all requests are collected, the thread releases the queue and sends the requests to the I/O interface. This request processing demands a nearly constant amount of

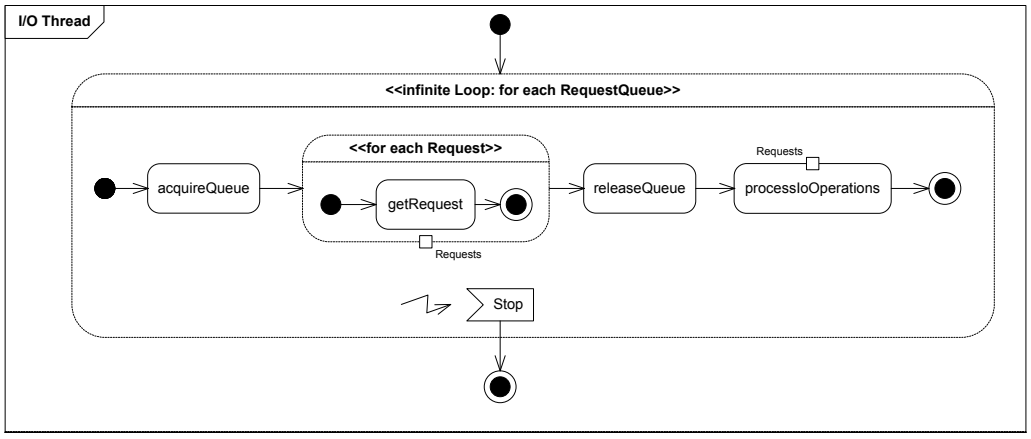


Figure 4.5: Activity diagram of an I/O thread

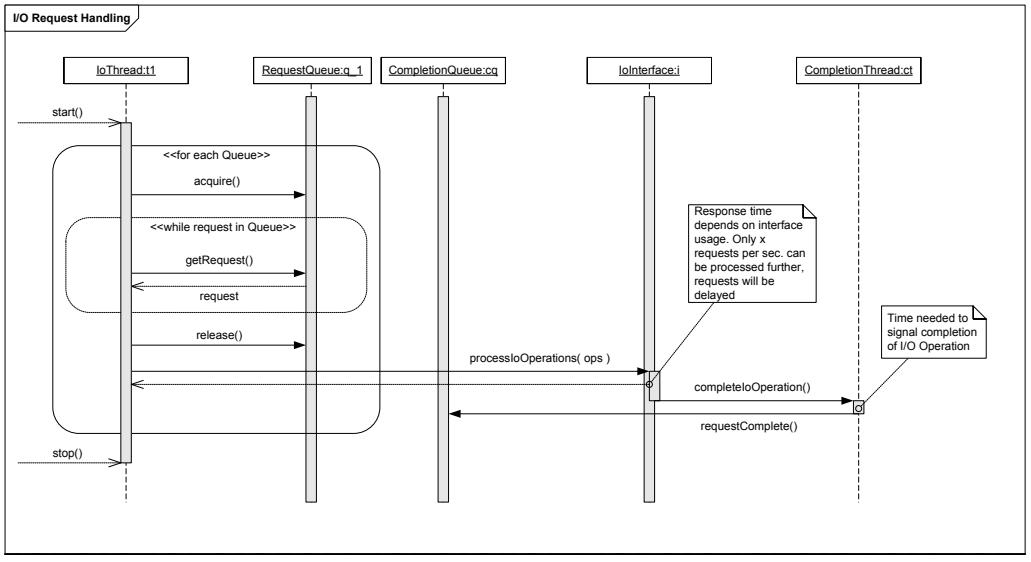


Figure 4.6: Request handling sequence diagram

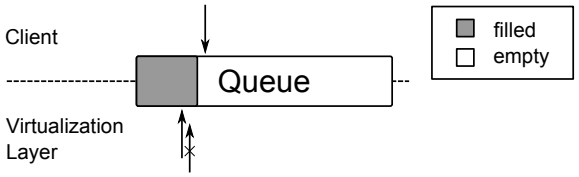


Figure 4.7: Queue locking scenario

CPU time as it simply copies the pointer to the memory area, not the request itself. In contrast to the queues, it is possible to concurrently access to the I/O interface and each thread can hand over its requests without being blocked.

Whenever a request is passed to the I/O interface, the operating system of the VL providing the I/O interface processes and executes the I/O operation by sending it to the storage hardware. As in the synchronous case, the delay from the arrival of a request until the storage hardware signals the completion is composed of the time spent in the I/O interface's and the storage hardware response time.

At last, when the completion of a request is signaled by the storage hardware, a completion thread is started. This thread accesses the completion queue of the corresponding request queue the request was issued. It puts the results given by the storage hardware into the completion queue. Now, the client can check the completion queue for new signals and then he will notice which requests completed.

Furthermore, the more requests are issued from the same request queue, the more likely it is that a completion thread is blocked. Hence, it is delayed if it wants to put the completion signal in the corresponding completion queue. Additionally, the completion thread competes for the VL's CPU like the other CPU-using components like I/O thread and I/O interface. Thus it could have an impact on the overall system performance.

4.3 Performance relevant parameters and system behavior

This section discusses which parts of the presented architecture should be explicitly modeled because of their possible influences on the system's performance. At first, the performance influences of possible blocked queue accesses will be explained. After that, further parameters and their dependencies considered to be performance relevant will be discussed.

4.3.1 Queue access and blocking

Given the architectures described in the last subsections, it is intuitive that blocking of threads and thus delays will occur. These resulting delays are expected to have an impact on the overall system performance, especially if the number of concurrent threads is increased. Therefore, the understanding of why and how threads are blocked is very important to create a proper model of this part of the system. This subsection explains the queue access and discusses queue blocking effects in detail.

Figure 4.7 gives a schematic overview of the queue's design. The design enables that client and VL can work on the queue, concurrently. However, if one thread of the

configurable	variable
runtime of threads	amount of CPUs
runtime of IO interface	CPU power
request type	amount of IO threads
request size	IO interface throughput
storage hardware delay	amount of request queues
queue access time	

Table 4.1: Classification of performance parameters

VL accesses the queue, it locks the queue to prevent accesses of other VL threads. Accesses from the client will not be blocked. To enable this, there exist two areas inside the queue marked by e.g. pointers. The gray area of the queue is the part which is filled with requests and can be accessed by only one VL thread. The white area of the queue is empty and can be accessed by only one client thread to put new requests into the queue. These areas can be marked by pointers which will prevent client and I/O thread from blocking each other.

In the asynchronous scenario, there are two queues of the mentioned design. The first one is the request queue of the Request Completion Queue Pair. When an I/O thread tries to access a request queue, it is possible that another I/O thread is already working on this queue and has locked it. In this case, the blocked I/O thread is delayed until the request queue is unlocked by the thread which locked the queue. However, the special design of the request-completion queues (see fig. 4.7) prevents clients from blocking I/O threads and vice versa. Hence, if an I/O thread accesses the request queue while a request is put into the queue by the client, the access is not blocked. The case when two requests are put into the request queue by the client is unspecified because it has no influence on the performance on the VL.

The second shared queue where conflicts can occur is the completion queue. Analog to the last paragraph, a completion thread is blocked if it wants to access a completion queue and this queue is already locked by another completion thread. This case may happen if two requests complete at almost the same time and their completion signal must be delivered to the same completion queue at the same time. Then, the first of the two completion threads blocks the access of the second thread. Likewise for the request queue, the completion thread is not blocked if a client reads the completion signal in the completion queue.

4.3.2 Performance parameters

The parameters of the here created performance model can be categorized into “configurable” or “variable” parameters (table 4.1). Variable parameters are integrated into the model such that the user of the model can change them. This enables the model user to observe the performance behavior of the model with different parameter settings.

In contrast, configurable parameters are the parameters of the performance model which can be changed, but should not. They serve as possibilities to configure and fine-tune the performance model until it matches the behavior of the reference system. This configuration will be explained in detail in chapter 6.

4.3.3 Variable parameters

One of the main variable parameter influencing the system is the amount of request queues. The amount of client's served by the VL is not fixed and can change while the VL is running. As each client is configured with at least one device, the amount of request queues increases with the an increasing number of clients. Furthermore, each client can be individually configured with more than one device which increases the amount of request queues, additionally. The amount of request queues could have an influence on the system performance as the amount of request queues changes according to the amount of clients and devices. One can imagine that if the amount of request queues decreases to a very low level, the probability that an I/O thread is delayed because of queue locking increases. The amount of request queues is classified as "variable" because the model user should be able to observe the performance impact of a varying amount of request queues with a varying amount of threads.

This is why another variable parameter is the amount of I/O threads. It is an intuitive assumption that varying this thread pool size can have a performance impact on the system, because an increasing number of I/O threads could either increase the overall system throughput or it could delay other I/O threads because they block each other. As mentioned in section 4.1, it should be possible to change the amount of CPU's assigned to the VL and therefore the CPU power that is available to the VL. It is obvious, the amount of CPUs and thus, the overall CPU power must influence the system performance.

Another parameter influencing the system performance is the throughput constraint of the I/O interface. It only offers a specific amount of requests to be handled in a certain unit of time. If the I/O interface becomes a bottleneck because its capacity is insufficient, it is possible to assign additional capacity to the I/O interface which might increase the throughput of the whole system. Hence, the I/O interface throughput is categorized as variable, too.

4.3.4 Configurable Parameters

The queue access time is considered to be performance relevant, because it influences the amount of time it takes a thread to access a queue which in turn might have an influence on the throughput. Primarily, this queue access time will depend on the variable parameters amount of queues and amount of I/O threads. In case of the completion thread, the queue access time will more depend on the rate, the requests arrive and complete. Hence, this configurable parameter queue access time should not be set manually, it should result from the model's design.

In a real world scenario, one has no influence on the type and the size of the requests a client issues. Hence, request type and size are classified as configurable parameters, though the user will have to change them to simulate different scenarios. Both, the type and size of a request can have a performance impact. This results of the idea that one request type (e.g. WRITE) might need more CPU time to be processed or is delayed longer as READ. The second argument is that an increasing request size might lead to an increasing hardware response time.

This storage hardware response time and the runtime of I/O thread and completion thread are not variable in reality. They could only be influenced by changing the implementation of the threads. Hence, their runtime is considered to be fixed within

the model, too. Consequently, the runtime of I/O thread and completion thread must be measured and then put into the model. The runtime of the I/O interface depends on request size and type and is not variable, too.

5. Model implementation

This chapter describes the modeling of the architecture introduced in chapter 4. Because PCM was primarily targeted on business information systems, some assumptions and limitations of PCM complicate the usage of PCM in the specific domain of this work. Hence, some components of the architecture had do be modeled in a different way, because they could not be mapped directly to elements of the PCM metamodel. Nevertheless, the realization of this performance model of the architecture should be accomplished in the most accurate manner.

This chapter focuses on creating a model skeleton, whereas the following chapter 6 will explain the configuration of the model. The problems, assumptions and solutions to create this model are discussed in the first section 5.1, followed by section 5.2 which gives a detailed explanation of the model. Finally, section 5.3 summarizes the complete modeling approach.

5.1 Limitations and Assumptions

PCM was primarily developed to analyze software architectures of business information systems. As I/O virtualization is a different domain, some challenges and limitations arose which complicated the modeling of the architecture described in chapter 4. This section discusses the limitations of PCM and how the resulting problems can be solved. Furthermore, the validity of the assumptions made to solve these problems will be discussed.

5.1.1 Challenges, limitations and solution patterns

As already emphasized, PCM focuses on modeling and analyzing the software architecture of business information systems. During the development of PCM, several limitations arose and assumptions were made. They are detailed described in [Bec08, Koz08]. However, some of these assumptions are invalid in PCM-foreign domains like this work, e.g. the abstraction from state. Further problems arise because of the special requirements which are different from the requirements of business information systems. However, some of PCM's limitations pose problems in general, too. Other works [And08, KR08] already faced some of these difficulties.

However, they had to invent different solutions to the one presented in the following because of different requirements.

At first, the limitations of PCM and the resulting model-specific problems will be listed. They are followed by a description of technical problems and limitations which occurred during this work. Each problem is supplemented by a preferably general solution pattern, if applicable, or a work-around which overcomes the shortcomings in a problem-specific manner.

5.1.1.1 Limitations of PCM

The fundamental limitations of PCM influencing this work are listed in the following.

- **Static architecture**: The modeled architecture is static. Neither the connectors of components can change, nor can components move to different hardware resources. This implies that it is not possible to have a varying amount of component instances during analysis or simulation. This makes it extremely hard to model components which should have several instances, e.g. request or completion queues.
- **Abstraction from state**: PCM does not regard the internal state of components or the runtime environment. Hence, it is not possible to adapt the component's behavior at runtime. For example, it is impossible that one component can store the information which queues are locked or which component has locked queues. In this work, this makes a straight-forward modeling of queue locking impossible.
- **Active components**: There are no active components in PCM. Each component is triggered by method invocations. There is no component type which can actively call other components autonomously. Hence, modeling an I/O thread which actively polls the request queues is impossible out of the box.
- **Scheduler support**: Currently PCM only supports abstractions of scheduling strategies like processor sharing or first-come-first-served and single processor scheduling only. Furthermore, there is no support of multi-core platforms.

The following describes the difficulties arising from the previously listed limitations. Each problem is described in a general and then work-specific manner, also its solution.

Active Components

The only active part within a PCM model is a *UsageBehavior*. Hence, components are called by e.g. such a UsageBehavior or other components and cannot autonomously call other components. Currently, components are not intended to actively do something. However, such components could be useful to model e.g. the frequent persisting of data. In this work, an active component is required to model the behavior of the I/O thread. Recall that the I/O thread actively polls the request queues for new request and processes any requests stored in the queues.

Solution: This active behavior can be modeled by an additional provided interface. This interface allows to trigger the component from the outside of the system in

a usage scenario. In the following model, the I/O thread provides such a trigger-interface and the usage scenario of the usage model calls this interface with a specific rate. One can then use the two workloads types to model different behavior. For example, a closed workload realizes a continuous trigger without concurrency, whereas an open workload with an interarrival time employs a behavior repeating with a special frequency.

Capacity constraint

Another general challenge is how to model a specific resource-independent capacity constraint of a component, e.g. a throughput of requests independent of the request's size. In PCM there are no components or other possibilities to model such a behavior without additional effort. Basically, this challenge arises from the fact that components do not have internal states and thus cannot count e.g. the throughput. Using a simple passive resource with a capacity corresponding to the capacity which shall be modeled is no solution. It would only solve the problem that n processes can use this passive resource in parallel, independent of the time. Hence, it cannot restrict the amount of parallel processes per time.

Using an active resource like a CPU is not possible for the following reasons. The major reason is that the active resource cannot be limited as there is no proper scheduling policy meeting this special requirement of capacity restriction. In case of `PROCESSOR_SHARING`, there is no possibility to limit the number of parallel executions as well as with `DELAY`. Another problem of each scheduling policy and an active resource in general is that a request would be delayed even if there was enough capacity available.

Another possible solution one could think of is to combine a passive resource with an active trigger, used in a dedicated usage scenario. This trigger activates a dedicated component to increase or decrease the capacity of the passive resource. But this approach has the following disadvantage. Assume that the passive resource is decreased less often than the trigger of the usage scenario calls increase or vice versa. Then, this would lead to a raise of the passive resource's capacity above the limit which was originally set.

Solution: A general solution pattern for a capacity constraining component is the *CapacityController*. It provides an interface to be accessed and contains a passive resource. The passive resource's capacity corresponds to the maximum capacity restriction the real component has. To model the maximum parallel access per time, the SEFF of the provided interface must implement the behavior depicted in figure 5.1. At first, the passive resource must be acquired. If no passive resource is available, this call of the provided interface will be delayed until capacity is available. In case of available passive resources, the control flow is forked by the following *AsynchronousForkAction*. This asynchronous fork is necessary, as the original control flow can continue and return. The second forked control flow contains an *InternalAction* which delays the control flow for the specified amount of time. After this delay, the acquired passive resource is released and the SEFF is complete. For example, if there is a component with a throughput constraint of 10 per second, the passive resource's capacity of the model component would be set to 10. The resource demand DELAY would be set to one second. Hence, any acquired passive resource would be released after one second.

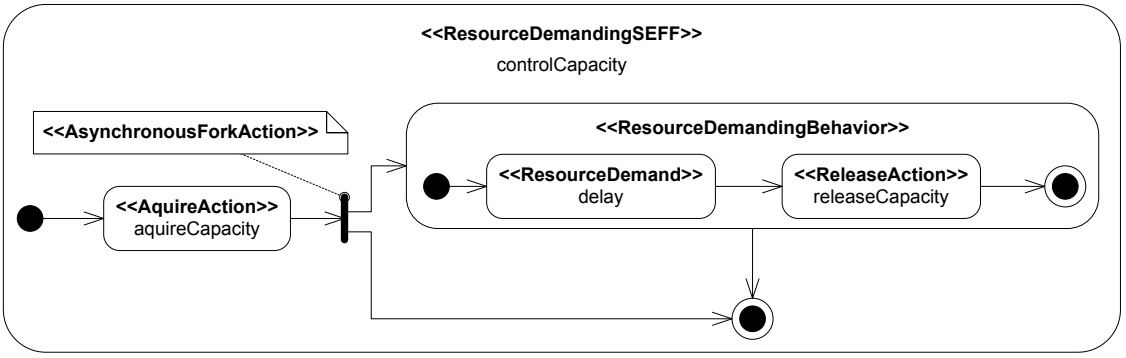


Figure 5.1: Resource Demand Service Effect Specification of the *CapacityController's* internal behavior

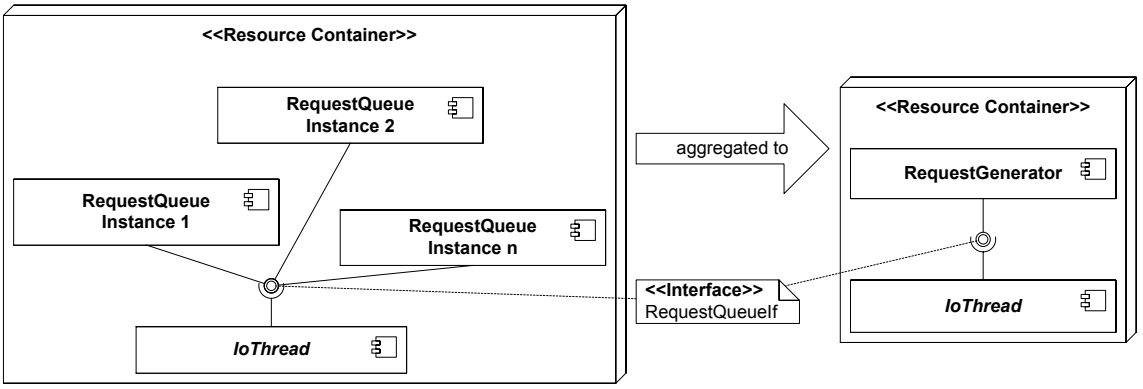


Figure 5.2: Aggregation of several queues to one component

Returning to the VL architecture model, it is possible to ensure the I/O interface's throughput constraint by using this *CapacityController* and connecting it to the I/O interface. The realization of this within the model is explained in section 5.2.

Multiple component instances

In general, with PCM there are no restrictions preventing from modeling a specific amount of components, interconnected with a different component. Figure 5.2 gives a short example how a several instances of the same component type can be connected. But if this amount of component changes, the model must be manually adjusted by adding or removing the components one by one. There is no possibility in PCM to specify that a specific component should have several instances. This shortcoming could be solved with inplace transformations which can automatically generate components and connect them to others. However, implementing such a transformation would go beyond the scope of this work.

Transferring the described problem to this work, the question remains how to model a varying amount of several request queue and completion queue components respectively. For the software architect, it is possible to allocate several different instances of the queue model component to the resource container, but only with tremendous effort, because each queue component must be allocated and connected explicitly (see figure 5.2). Assuming that creating and connecting a variable amount of components is possible, it would still be hard to specify and especially maintain the access of the multiple instances, e.g in figure 5.2 to direct the thread component

to the queue component it has to access. This could be achieved by branches but would cause enormous effort to explicitly model this circumstance. Especially, if the parameters are changing, one would have to adjust these branches.

For this work, a one-to-one modeling of the request queues accessed by I/O threads and the completion threads and queues would be too much modeling effort. Especially because the amount of request queue components is a variable parameter and changes frequently. Moreover, it is undefined how an I/O thread component shall access the request queue components which implies it is not possible to specify or model an order in which e.g. the thread component has to access each queue component. Apart from the tremendous modeling effort, an explicit model would most likely not lead to a more accurate result as the solution presented in the following.

Solution: Because of the previously explained problems, a solution to model several instances of a component is to combine them into one component. The accesses of a specific component must then be reflected by probability distribution calculations in branches and other SEFF-actions. Concerning the problem of how to model several request queues, all instances were aggregated in one component. This queue-representing component must meet two requirements. At first, it is responsible for the request generation, because a queue access shall result in a returned request. Second, it must reflect the behavior of threads which access and lock queues and hence block each other. These blocking probabilities and resulting delay times are now calculated by probability functions, using the stochastic expressions PCM supports. How this delay time and the blocking probabilities are calculated is explained in section 5.1.2.

After a request completes and the completion thread has to signal the request completion, exactly the same problem arises. However, this time the queue access and blocking must be based on more assumptions then for the requests queues and I/O threads. Because of these assumptions, the same solution is not adequate and thus, completion queue locking is modeled by simple approximations and probability distributions presented in section 5.1.2.

5.1.1.2 Technical limitations

The simulation engine of PCM currently works with native Java threads, i.e. for each event created by the usage scenario, a new thread is started. Moreover, *AsynchronousForkActions* fork new native threads, too.

In the scenarios of this work, a request arrival rate of over 50000 requests per second is not exceptional. However, such scenarios cannot be simulated without increasing the Java VM memory size and decreasing the stack size of Java threads. Otherwise, OutOfMemory-exceptions are likely to occur when running the simulation, especially for a lot of users. Hence, to run the simulation without errors, the Eclipse PCM Bench 3.0 must be started with the following additional entries in eclipse.ini:

```
--launcher.XXMaxPermSize320m
-vmargs -Xss8k -Xms512m -Xmx1024m
--XX:PermSize=320M
```

The described problem usually is no limitation within the target domain of PCM as they generally do not produce such an high amount of concurrency. Nevertheless, it

is a problem and constrains the potential of PCM in this domain. Although it can be avoided by adjusting the Java VM parameters (heap space, stack size, etc.) a better solution should be provided and is currently in work.

5.1.2 Assumptions

This section presents the assumptions made during the modeling process. These were necessary to model or represent special circumstances of this work at all. Especially the delay resulting from queue access delays could have an impact on the system performance. Hence, these influences shall be captured by the model but can only be reflected with approximations, probability distributions and/or assumptions. The following is divided into the assumptions made to model request queue locking and completion queue locking.

Request queues

Whenever an I/O thread tries to access a request queue, the queue might be locked by another I/O thread. Then, the current I/O thread will be delayed. Because of PCMs shortcoming of modeling a variable, parameterizable amount of queues, the queue locking and delay of threads is approximated by probability calculations. As the request queue access order is unspecified, it will be assumed that this access order is randomly distributed. Thus, the queue access and resulting delays can be approximated by a special function calculating the delay of a thread. The queue access delay is calculated depending on the amount of I/O threads t, the number of request queues q and the average delay s of a non-blocked queue access. The determination of the thread delay is visualized by the activity diagram of figure 5.3.

At first, the average blocking probability m of a thread t when accessing one specific queue has to be calculated. The probability that thread t accesses exactly one out of q queues is $p = \frac{1}{q}$. Hence, the probability that another thread accesses exactly this queue is the complementary probability $\overline{p} = (1 - p)$. Furthermore, the probability that one of n other treads accesses exactly this queue is $\overline{p}^n = (1 - p)^n$. Hence, the probability that one of n threads does not access exactly this queue is again the complementary probability $1 - \overline{p}^n$. Assuming that one specific thread tries to access one specific queue of a system with q queues and $n = t - 1$ remaining threads, its blocking probability is

$$m(t, q) = 1 - (1 - \frac{1}{q})^{t-1} \tag{5.1}$$

Furthermore, the average blocking probability $m(t, q)$ is used to calculate whether a thread is blocked or not. This is achieved by using a binomial distribution function *Binom(size,probability)* (see section 2.4.7) with the parameter values $size = 1$ and $probability = m(t, q)$. Then the call of $Binom(1, m)$ returns the value 1 with the expectation value of $m(t, q)$. In this case, 1 means that the thread was blocked, 0 means not blocked.

If a request queue is accessed and the thread is not blocked, the thread will be delayed by s. If the queue is locked, a Poisson distributed ($Pois(rate)$, see section 2.4.7) random deviate n is calculated. For example, if the blocking probability is $m = 0.05$,

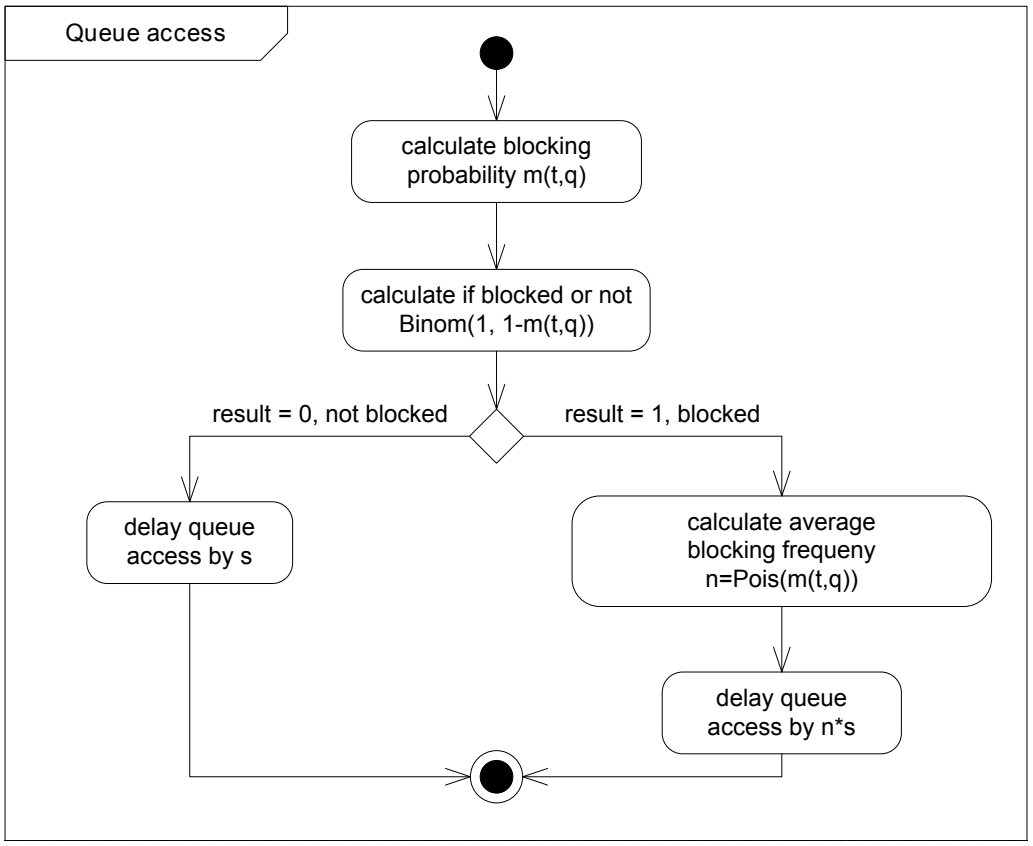


Figure 5.3: Calculation of the I/O thread queue access delay

5% of all queue accesses are blocked. This means, each twentieth access results in a blocked and repeated access. How many consecutive tries will be unsuccessful is defined by the Poisson distribution. An example for this distribution function was given in figure 2.6a. In the case of request queue accesses, $n = Pois(m(t,q))$ reflects the amount of blocked queue accesses. In case the thread is blocked, it will block n times and therefore be delayed by $n \cdot s$. The rate-parameter of the Poisson distribution is set to $rate = m(t,q)$, the average probability that an I/O thread is blocked because another thread already locked this queue. Formula (5.2) subsumes the previous explained parts.

$$delay(s,t,q) = s \cdot [1 + Binom(1, m(t,q)) \cdot Pois(m(t,q))] \tag{5.2}$$

Completion queues

The completion thread working on the completion queues to signal the result of a request faces the same challenges as the I/O thread. Recall that PCM does not support a dynamic structure where the completion queues can be generated automatically at runtime. Hence, there must be an equivalent solution with probability distributions comparable to the *RequestGenerator* component. However, in this case the major problem is that it is unknown how much requests were issued, how much are still in the system, when they complete and which client issued the requests. Because of these unknown variables, the same calculation as for request queues seems inappropriate.

Assuming an equally distribution of requests per client is not a valid assumption, as it is not true that all requests processed by the I/O interface were issued by all clients equally. In reality, a scenario where some clients issue most of the requests is more reasonable. Therefore a blocked completion thread is much more likely than assuming an equal distribution of completion signals over all completion queues. Therefore it is assumed that a simple estimation of the blocking probability and access delay does not have worse results than an equation with to many unknowns. Hence, the completion thread delay caused by blocked queue accesses will be based on the following simple calculation, assuming a standard access delay of s.

$$delay(s, p) = s \cdot [1 + \cdot Binom(1, p) \cdot Pois(p)] \tag{5.3}$$

Both the I/O and completion thread, the determination of the actual queue access time s would be too costly to measure. Hence it is estimated as a percentage of the corresponding thread runtime. In case of the I/O thread, this percentage is set to 5% whereas it is set to 10% for the completion thread. As these values are only estimations, a calibration might be required which will be discussed in chapter 6.

5.2 The model implementation

As mentioned before, this section explains the model according to the architecture of chapter 4. First of all, the next subsection gives a coarse overview of the system view with the modeled components, their component parameters and interfaces. This is a followed by a detailed explanation of the model skeleton and the remaining parts to complete the PCM model instance, the usage model and the resource environment. The important part, the parameterization of the model, is explained in subsection 5.2.3 and deals with the variable parameters the model user can change to analyze the behavior. The missing resource demands of the here created model skeleton will be supplied in chapter 6.

5.2.1 Model overview

In the following, both *IoThread* and *IoInterface* represent abstract components. PCM does not support abstract components like the ones depicted in figure 5.4, but the figure demonstrates that they can be easily replaced by two different implementation types (synchronous or asynchronous version). The synchronous version will be used to calibrate the model according to the real system. After the model reflects the behavior of the reality, the synchronous components can be easily replaced by their asynchronous counterpart. This flexibility in exchanging components demonstrates the advantage of PCM.

To interact with the system, it provides one interface, namely the *IoThreadIf* interface. This is the trigger for the *IoThread* component to activate the request processing. The abstract *IoThread* component represents a concrete thread type which collects requests from the request queues. It must be replaced by either the asynchronous or synchronous implementation explained in chapter 4. Abstract components in figure 5.4 are depicted by italic names. In the synchronous case, the synchronous *IoThread* waits for the control flow to return from the synchronous implementation of the *IoInterface*. In case of the asynchronous implementation of the

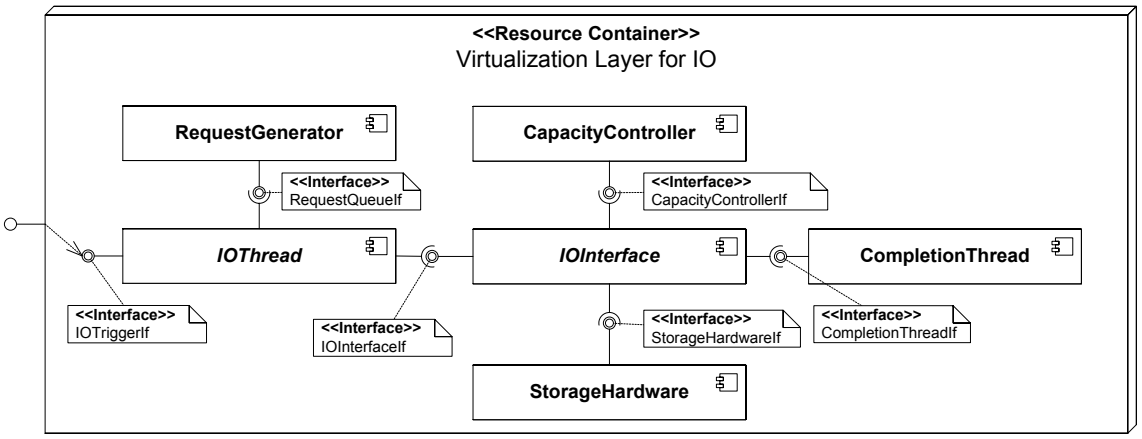


Figure 5.4: System diagram with abstract components *IoThread* and *IoInterface*

IoThread, the control flow is forked. One part remains within the system and continues to the asynchronous *IoInterface* and the other part (the call of the asynchronous *IoThread*) returns to handle the next request.

Starting at the *IoThread* component, the control flow transits to the *RequestGenerator* component via the *RequestQueueIf* interface. Because of the special limitations when modeling the request queues (see section 5.1.1), all request queues are aggregated in this *RequestGenerator* which implements the interface. Compared to the VL architecture, the *RequestGenerator* is the representation of all request queues. Moreover, clients are not explicitly modeled, as attaching new clients only results in an increased amount of request queues. This amount of request queues can be set as a component parameter of the *RequestGenerator*.

Furthermore, each call of the *RequestQueueIf* interface will be delayed by the *RequestGenerator* depending on the parameters *IoThreads, RequestQueues* and *IOThreadRuntime*. These parameters and their influence on the model behavior will be explained in the section 5.2.3. After this delay, the *RequestGenerator* generates and returns a request according to parameters which are specified within the *RequestGenerator*'s RD-SEFF.

Once the generated request is returned by the *RequestGenerator*, the control flow continues from the *IoThread* and passes the request to the *IoInterface* via the *IoInterfaceIf* interface. The *IoInterface* component is abstract, too, and must be replaced by either synchronous or asynchronous implementation. In either case, the control flow continues to the *CapacityController* component by using the *CapacityControllerIf* interface. If capacity is available, the control flow returns immediately. If not, it is delayed until capacity is available.

Then, the control flow passes to the *StorageHardware* component via the *StorageHardwareIf* interface. This component models the storage hardware described in the architecture chapter. Depending on the request type and size, the control flow will be delayed and then returned to the *IoInterface*. In the synchronous case the control flow returns back through all components, indicating the completion of the request. In case of the asynchronous *IoInterface*, the *CompletionThread* component is called which has to signal the completion of the request. When this call returns, the control flow of the asynchronous request handling is completed.

Property name	Description
Type	String value $t \in \{\text{READ,WRITE}\}$, specifying the type of request.
Size	Integer value $s \in \{1..\infty\}$, specifying the size of the issued request in kilobyte.

Table 5.1: Data-type "request" and its properties

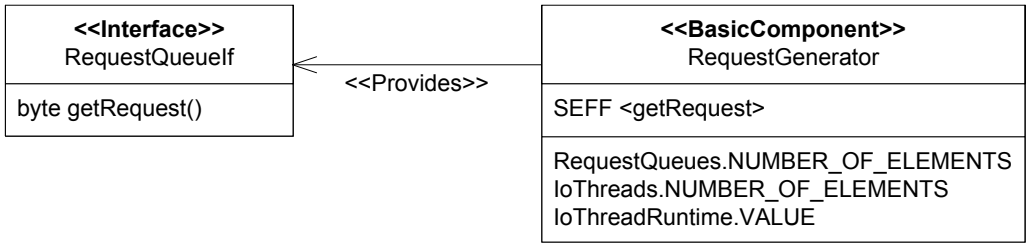


Figure 5.5: *RequestGenerator* component, representing the request queues

Thus, the difference between synchronous and asynchronous implementation of the *IoInterface* component is the call of the *CompletionThread* component in case of the latter implementation.

5.2.2 Component, resource environment, allocation and usage model description

The following sections describe the data-type used in the model as well as the modeled components, their implementation and hence their behavior in more detail. Furthermore, resource environment, allocation and usage model will be explained to complete the description of the PCM model instance. The resource demands of the components and their parameterization will be explained in the next section 5.2.3.

5.2.2.1 Data-type request

In this work, an I/O request has two special performance relevant properties, namely the request size and the request type. The values of these properties are specified in table 5.1. Because PCM allows to specify properties like `BYTESIZE` and `TYPE` for a primitive data type, I/O requests can be modeled by the primitive data-type `BYTE` which PCM supports out of the box. Both the properties TYPE and the `BYTESIZE` will be set in the *RequestGenerator* component according to values specified by the user. These properties are then used in the *IoInterface* and *StorageHardware* components to calculate CPU and DELAY resource demands.

5.2.2.2 RequestGenerator component

As already briefly described, the layer above the VL consisting of clients and request completion queue pairs (see figure 4.2) looks different in the model. Here, the request completion queue is divided in request and completion queue. The completion queue is accessed by the completion thread and will be explained later in this section.

The idea of the *RequestGenerator* (see figure 5.5) is to have one component which models the functionality of several request queues with the advantage that no generation tool is required. The *RequestGenerator* substitutes the request queues and

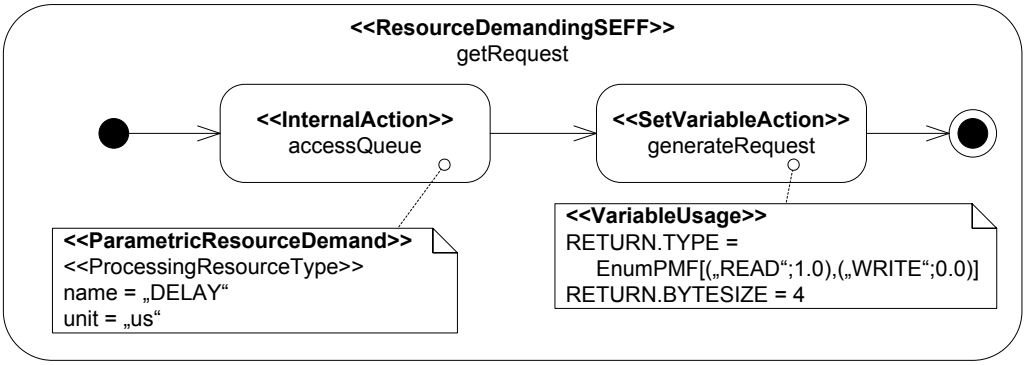


Figure 5.6: RD-SEFF of the *RequestGenerator*'s `getRequest`, configured to return 4KB READ requests.

thus provides the interface *RequestQueueIf* with the signature `byte getRequest()` to be accessed like a request queue.

The RD-SEFF of `getRequest` can be divided into two sections (see figure 5.6). First, `getRequest` is delayed by an *InternalAction*which requires $delay(s, q, t)$ units of the delay resource. This depends on the probability of being blocked and the standard delay when accessing a queue. The delay time is calculated depending on the component parameters: the number of I/O threads t, the amount of request queues q and the standard queue access time s. The calculation is described in detail in the subsection 5.1.2. Second, the request returned to the I/O thread is generated. The first part of the *SetVariableAction* sets the returned request type to READ (0), WRITE (1) or a mixture of both by using an integer probability mass function. Next, the size of the request is specified by setting the `BYTESIZE` value of the returned request.

5.2.2.3 I/O thread components

The synchronous or asynchronous *IoThread* component represents the set of all I/O threads of one type and requires two interfaces (see figure 5.7). The first is the *RequestQueueIf* interface which contains one signature `byte getRequest()`. The second is the *IoInterfaceIf* interface which is used to pass the requests to the I/O interface via `void processRequest(Request aRequest)`. Furthermore, the synchronous/asynchronous *IoThread* component provides an interface *IoThreadIf* which is used to trigger the I/O threads, because there are no active components in PCM (see section 5.1). The following explains the two different characteristics of the *Sync_IoThread* and *Async_IoThread* component.

Synchronous I/O thread

In the synchronous case, the behavior of the `handleRequest` RD-SEFF of the *Sync_IOThread* component is simple (see figure 5.8a). Every call of `handleRequest` directly leads leads to the *ExternalCallAction* `getRequest` of the *RequestQueueIf* interface to receive a request.

In the synchronous case, there are no restriction on how many I/O threads process the request and there is no delay because of blocked queues, as each thread works on

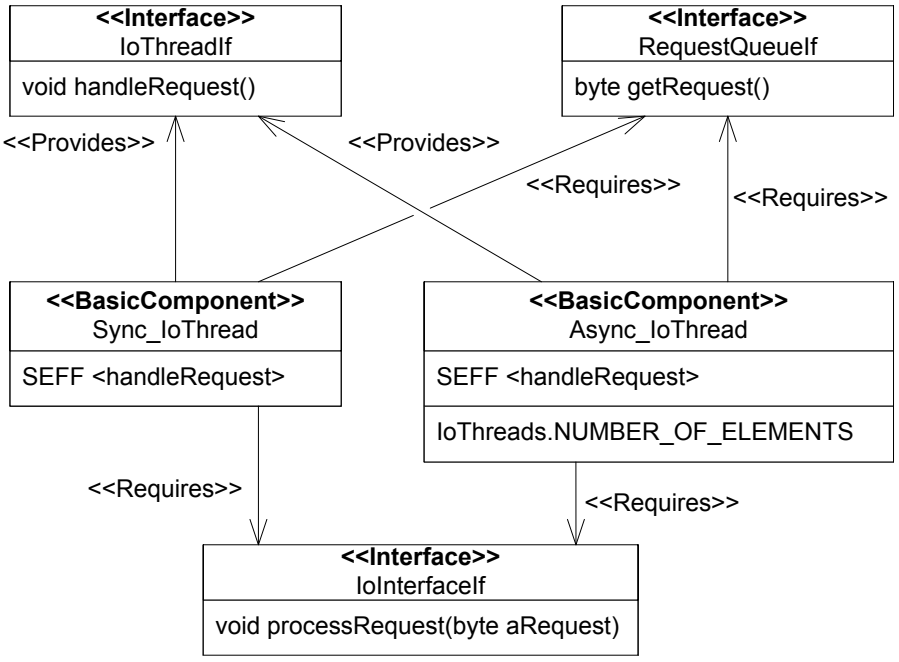


Figure 5.7: Synchronous and asynchronous I/O thread components with required and provided interfaces

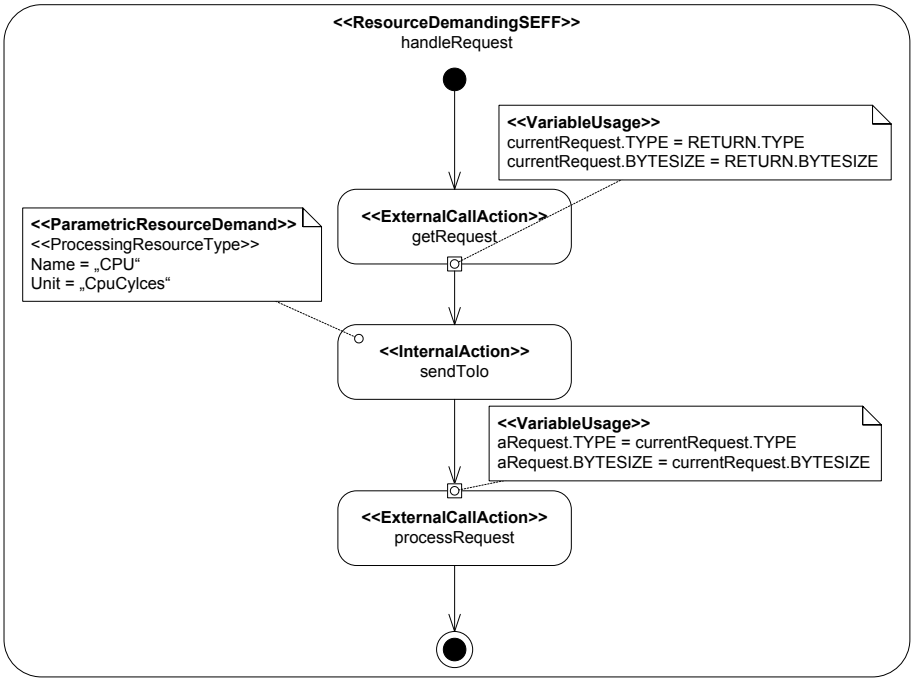
its own queue. After the call returns from the *RequestGenerator*, the *InternalAction* `sendToIo` requires a specific amount of CPU time which simulate the processing of request by the I/O thread. This resource demand is independent of any external parameter and only lasts a given amount of time, because the I/O thread only passes pointers to memory areas and does not copy or work on the requests (see section 4.1). How this configurable parameter is determined is explained in chapter 6.

Once the *InternalAction* completes, the request is passed to the I/O interface by the *ExternalCallAction* `processRequest`. The synchronous I/O thread must now wait until the call returns. Its job is not completed and the thread cannot handle further requests until this call returns. This behavior differs from the following implementation.

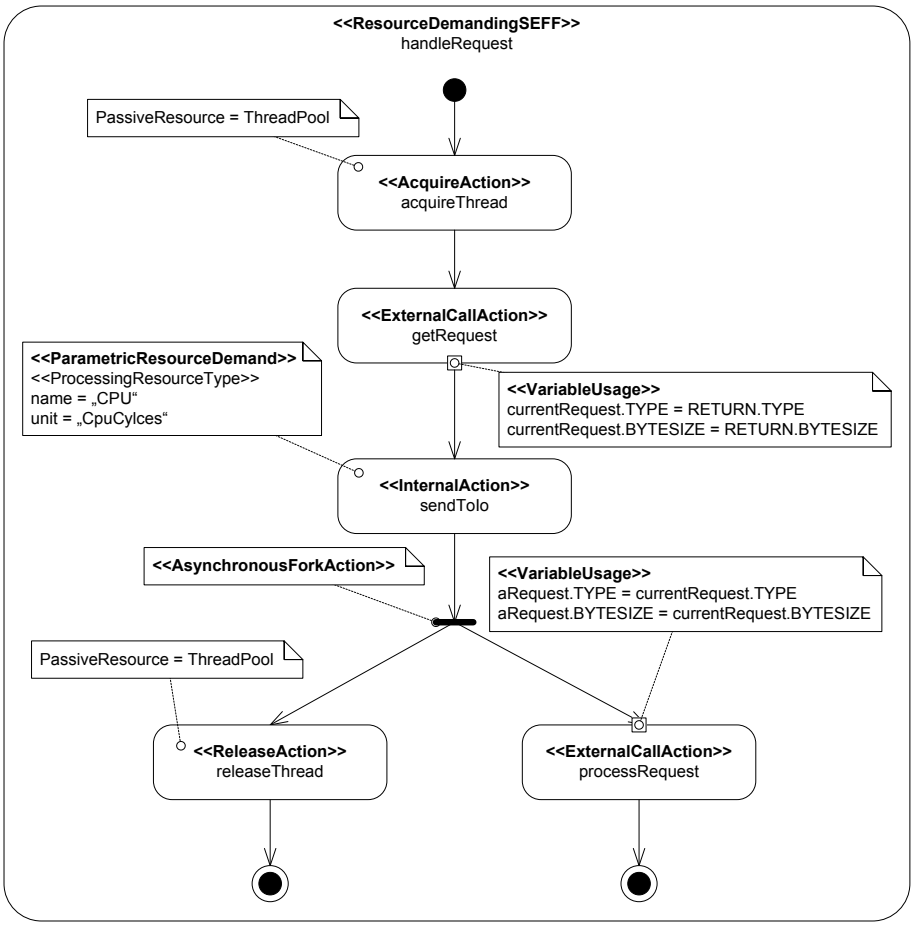
Asynchronous I/O thread

In the asynchronous case, the number of I/O threads processing requests from request queues is limited. How many I/O threads work on the request queues can be specified by the *Async_IoThread* component's parameter *IoThreads*. This specified `NUMBER_OF_ELEMENTS` sets the capacity of the passive resource *ThreadPoolSize* and hence delimits the amount of I/O threads working in parallel.

A diagram of the asynchronous `handleRequest` RD-SEFF is figure 5.8b. At first, the passive resource *ThreadPoolSize* is acquired to assure that only a limited number of threads is working on request processing. If the *AcquireAction* fails, the control flow is delayed until the acquisition is successful. The following steps are similar to the synchronous case. An *ExternalCallAction* of the *RequestQueue* interface to get a request. Depending on the parameter values of the *RequestGenerator* component, this call is delayed. After the *ExternalCallAction* returns, the *InternalAction sendToIO* represents the CPU resource demand of the IO thread. Again, this resource



(a) RD-SEFF of the *Sync_IoThread*'s `handleRequest`



(b) RD-SEFF of the *Async_IoThread*'s `handleRequest`

Figure 5.8: RD-SEFF of the synchronous (a) and asynchronous (b) I/O thread's `handleRequest`

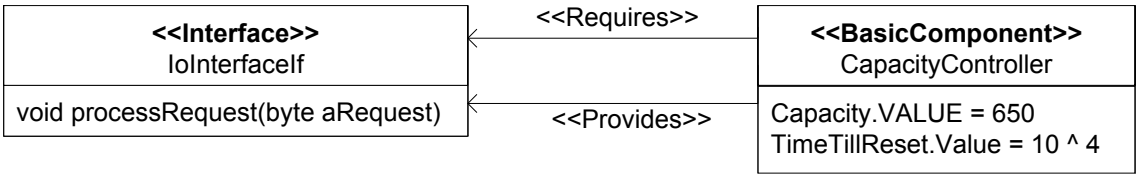


Figure 5.9: The *CapacityController* component

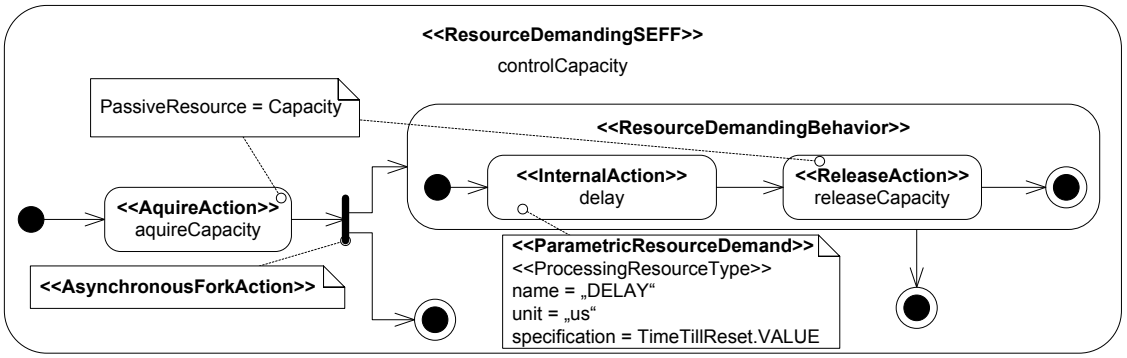


Figure 5.10: RD-SEFF of the *CapacityController*'s `processRequest`

demand is independent of any external parameters and only lasts a given amount of time. The parameter value will be determined in chapter 6.

Afterwards, the *ExternalCallAction* of `processRequest(byte aRequest)` passes the received request to the *IoInterface* component. In contrast to the synchronous case, this is done inside an *AsynchronousForkAction*, because the task of the I/O thread is completed now and the control flow can return to the user, simultaneously. Before that, the *ReleaseAction* must release the acquired passive resource so that another request can be processed by a new I/O thread. Hence, the thread pool is not modeled by n different component instances of the asynchronous I/O thread, but by the passive resource which limits the amount of simultaneously active control flows to the passive resource's capacity. This behavior is different to the synchronous case, where no such restriction is given.

5.2.2.4 Capacity controller

The *CapacityController* component (see figure 5.9) implements the solution pattern presented in 5.1.1.1 to address the problems of the I/O interface's throughput restriction. It provides the *CapacityControllerIf* interface to offer the *IoInterface* component its service of capacity controlling. The capacity available by the *CapacityController* is modeled by the passive resource *Capacity*. This capacity corresponds to the maximum available throughput of the I/O interface. The time after which the capacity is reset can be specified by setting the DELAY resource demand value of the `controlCapacity` RD-SEFF.

As already mentioned, the *CapacityController* component is used by the *IoInterface* by calling `controlCapacity`. Once activated, this RD-SEFF of the *CapacityController* tries to acquire the passive resource *Capacity* as depicted in figure 5.10. If no capacity is available, the control flow is delayed until passive resources are released. After the *AcquireAction*, an *AsynchronousForkAction* starts the release of the passive

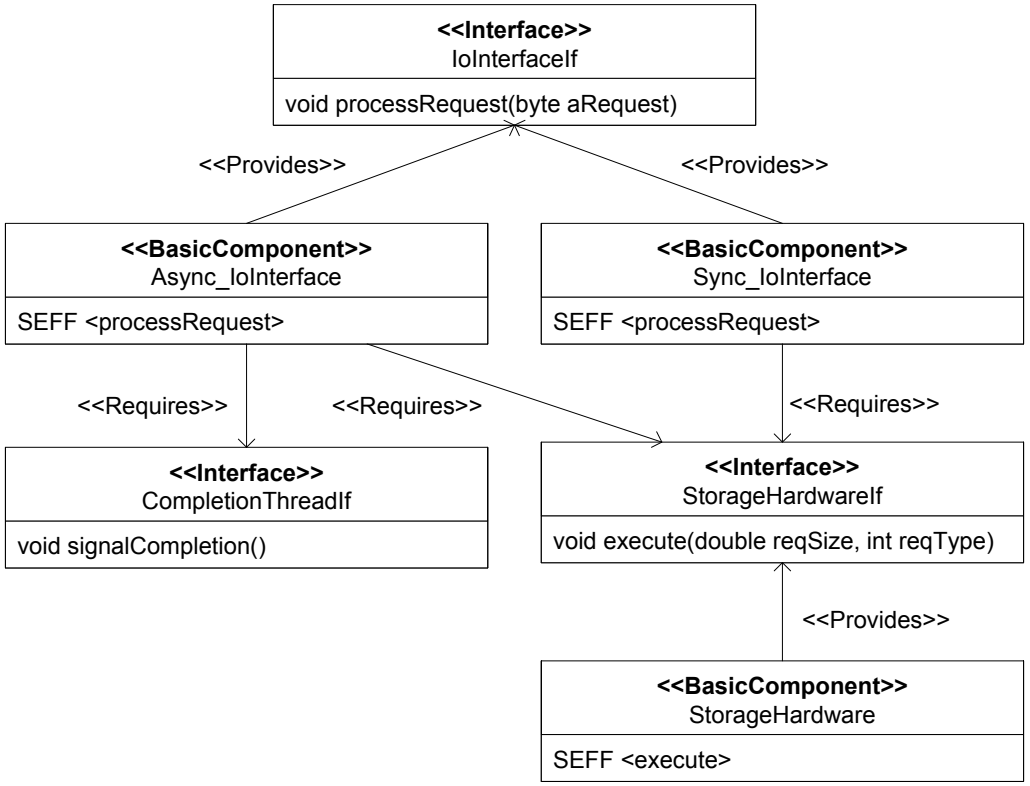


Figure 5.11: *IoInterface* and *StorageHardware* components with requires/provides interfaces

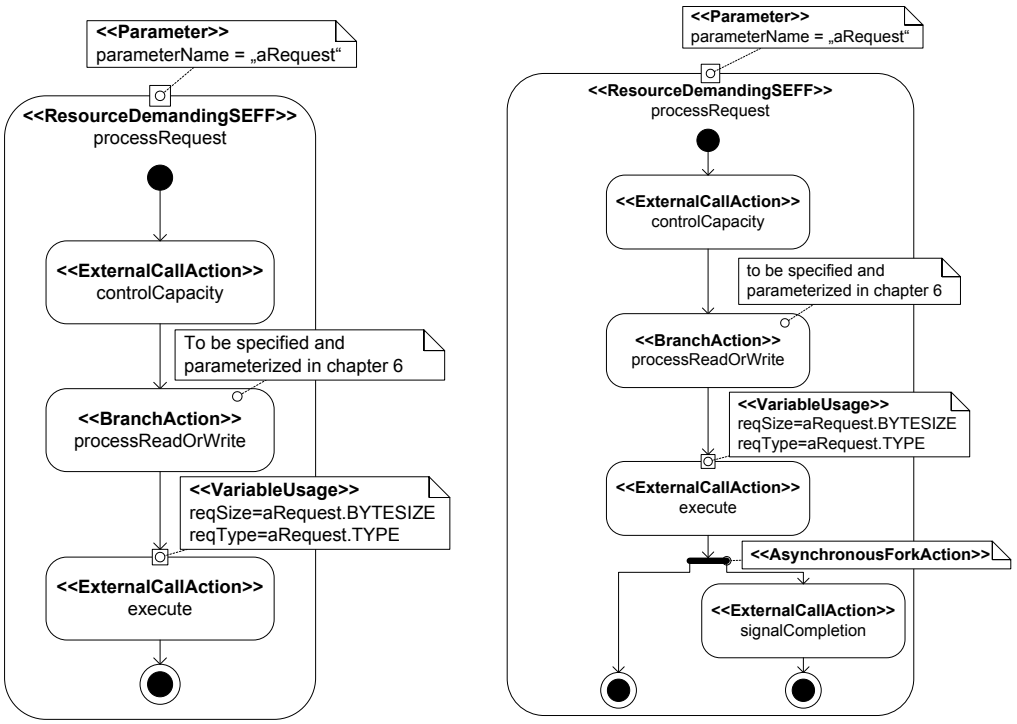
resource. This release is delayed by the *TimeTillReset* value specified as component parameter. While the *ReleaseAction* is forked, the call of `controlThroughput` can immediately return to the I/O interface and the *CapacityController*'s objective is accomplished. For example, if the throughput was 100 requests per second, the capacity of the *CapacityController* would be set to 100 and the *TimeTillReset* value would be set to one second. Hence, the release of the passive resource would take action exactly one second after it was acquired.

5.2.2.5 I/O interface and storage hardware

The I/O interface and the storage hardware as described in chapter 4 are modeled by two separate components (see figure 5.11). The first is the *IoInterface* component, modeling the software or operating system part of the request handling process. The second is the *StorageHardware* component, modeling the actual hardware with its delays.

The *IoInterface* component has one provides interface (*IoInterfaceIf*) with the signature `void processRequest(byte aRequest)` to receive requests from the I/O threads. Furthermore, it requires two different interfaces. The first is the *CompletionThread* interface to trigger the completion thread in the asynchronous case. Second, via `void execute(double reqSize, int reqType)` of the *StorageHardwareIf* interface, requests can be send to the *StorageHardware* component. This method call represents the actual execution of a READ/WRITE request on the storage hardware.

As for the *IoThread* component, the *IoInterface* component can be substituted either by a synchronous or an asynchronous implementation.



(a) RD-SEFF of the synchronous I/O interface's `processRequest`

(b) RD-SEFF of the asynchronous I/O interface's `processRequest`

Figure 5.12: RD-SEFF of the synchronous (a) and asynchronous (b) I/O interface's `processRequest`

Synchronous I/O interface

The *Sync_IoInterface* component receives the requests by the method signature `processRequest` of the provided interface (figure 5.12a). First, `controlCapacity` is called to assure that there is enough throughput available. If there is no capacity available, this call blocks until capacity is released. Then, a *GuardedBranchAction* checks, whether a READ or WRITE request has to processed. Depending on this request's type and size, the *IoInterface* demands resources within the *InternalActions* `processRead` or `processWrite`. The parameter values for these resource demands will be determined in chapter 6. Then, the request size and request type is passed to the *StorageHardware* component by the *ExternalCallaction* `execute`. In the synchronous case, the task of the *Sync_IoInterface* component is completed as soon as the call of the *StorageHardware* component returns and the control flow returns to the synchronous I/O thread.

Asynchronous I/O interface

In the asynchronous case, the RD-SEFF of `processRequest` is slightly different (figure 5.12b). Again, the execution of `controlCapacity` checks if the throughput constraint is maintained. After the resource demands are issued according to READ or WRITE requests, an *ExternalCallAction* calls `execute` and sends the request to the *StorageHardware* component.

The main difference to the synchronous case is that the `processRequest` RD-SEFF of the *Async_IoInterface* component has one final *ExternalCallAction*. It

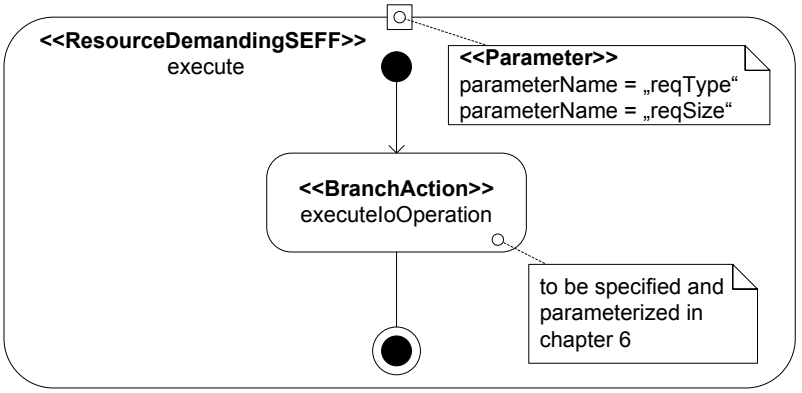
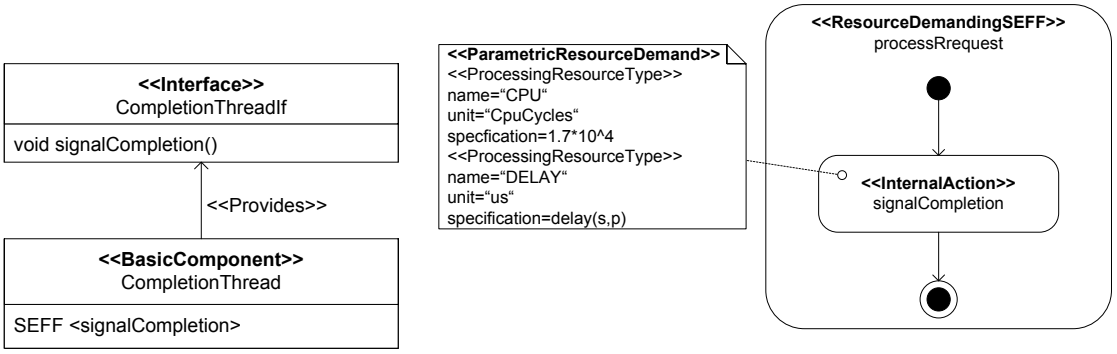


Figure 5.13: RD-SEFF of the *StorageHardware*'s `execute`



(a) (b)

Figure 5.14: The completion thread component (a) and its corresponding RD-SEFF `signalCompletion` (b)

calls `signalCompletion` of the *CompletionThread* component to activate the signaling of the request's completion. The workflow in the asynchronous case is not complete until `signalCompletion` is called. Like in the *Async_IoThread* component, this call is forked, as the control flow can return and does not have to wait for the `signalCompletion` call to finish.

5.2.2.6 Storage hardware

The RD-SEFF `execute` of the *StorageHardware* component is very simple (see figure 5.13). It consists of a *GuardedBranchAction*, checking the request type the RD-SEFF receives as a parameter. Depending on the request type, the resource demands are calculated and issued within the *InternalActions* `executeRead` and `executeWrite`.

5.2.2.7 Completion thread

The completion threads started by the I/O interface are modeled by the *CompletionThread* component. As this thread pool is not limited, this component contains no passive resource. The interface provided by this component is the *CompletionThreadIf* interface with the signature `void signalCompletion()`, used as a trigger to activate a completion thread.

The RD-SEFF of `signalCompletion` has only one *InternalAction* called *signalCompletion* which issues two resource demands. First, it requires CPU resources to process the information. Second, the completion thread might be delayed, because another completion thread already locked the queue, this completion thread wants to access. This case is not unlikely, because a client usually issues not only one request per device, but many.

5.2.2.8 Resource environment and allocation model

Before the modeled components can be deployed onto a special resource container, one has to define one or more corresponding resource container(s) in the resource environment. In this work, the whole VL runs on the same hardware. Hence, there is only one resource container called "*VL*", containing two processing resources, one is the CPU processing resource and the other is the DELAY processing resource.

The CPU processing resource corresponds to the CPU power provided by the CPU(s) assigned to the VL's partition. Its processing rate is specified in Hz. For example, if the VL runs on a 1.7 GHz CPU, the processing rate would be set to $1.7 \cdot 10^9$ Hz. As the current scheduling policies are not able to reconsider multi-core CPUs, adding more CPUs must be expressed by increasing the overall CPU processing rate. The scheduling policy of the processing resource must be set, too. In this work, `Processor_Sharing` (PS) is used.

Increasing the amount of available CPU cores is only possible if the exact scheduler of Happe is used. This scheduler is currently integrated into a dedicated PCM workbench instance which allows to specify the amount of replicas of a CPU core via the properties of the active resource. Furthermore, it offers three additional scheduling policies, namely `LINUX26`, `WINDOWS2003` and `WINXP`. However, this scheduler is not fully integrated within the current PCM bench version. The configuration of a version currently under development will be explained in section 6.4.2 when the influences of available CPU cores are discussed.

The DELAY processing resource is used to model delays occurring e.g. if a thread is blocked or to simulate the execution of a request within the storage hardware.The delays specified in this work are specified in μs. To assist the component developer or software architect when entering delay values, the DELAY processing rate is set to a value of 10^6. This enables the declaration of delays as μs-values. For example, if a DELAY resource demand of $5\mu s$ should be issued, the entered value in the DELAY resource demand would be 5. As this value would be divided by the DELAY processing rate 10^6, the resulting delay would be $\frac{5}{10^6} s = 5\mu s$.

Now that both the system components and the resource environment are specified, one can allocate the components to resource containers. In this case, as there is only one resource container, all components are allocated to the *VL* resource container previously described.

5.2.2.9 Usage model

Finally, the usage model must be defined. It consists of only one usage scenario which contains one *SystemCallAction*, calling the system model's provided interface. This call is delegated to the provided interface of the *IoThread* component. Hence, it is the trigger for the active I/O thread to process requests. In which way and

frequency the trigger is released can be specified by the kind of workload. In the test scenarios and experiments conducted in the following parts of this work, a request producing test application will be used. This application can simulate the amount of applications which issue requests to the system. Each of these simulated applications can be compared to a user which issues a request and waits for the results to return. Then, the next request is issued.

This behavior corresponds to the *ClosedWorkload*, where the value *population* specifies the amount of users (here the amount of simulated applications). To avoid that all requests are simultaneously issued over and over again, the *ThinkTime* of the *ClosedWorkload* was set to `UniInt(0,9) * 10 ^ -6`. This models a small variance (between 0 and $9\mu s$) the time spent until the next request is issued.

In the asynchronous scenario, such a *ClosedWorkload* might not be the best choice, as the almost immediate return of the asynchronous I/O thread might lead to a always fully loaded system. An *OpenWorkload* with a specific *InterarrivalTime* can be a solution to simulate little load. This problem will be further discussed in chapter 7.

5.2.3 Component parameterization

The software architect designing the system can overwrite all component parameters (e.g. the thread pool size) by setting new values in the properties view of a component. Furthermore, component parameters ease the usage of the model as detailed knowledge of the component's implementation is not required. Table 5.2 gives an overview of the available component parameters the system architect can change. For example, The component parameters *IoThreads* and *RequestQueues* are variable parameters as defined and explained in section 4.3. Thus, they are meant to be changed to observe the model's behavior. The following describes these variable parameters and how they can be used to influence the system's behavior. Furthermore, all other configurable parameters which do not require a detailed calibration will be explained, too. The configurable parameters requiring a detailed determination and calibration are described in the separate chapter 6.

5.2.3.1 RequestGenerator

Whenever an I/O thread tries to access a request queue, the queue might be locked by another I/O thread and the current I/O thread will be delayed. All the calculations presented in section 5.1.2 can be combined to one expression, calculating the delay of an I/O thread. This expression is used for the resource demand in the RDSEFF `getRequest`. The stochastic expression follows the mathematical expression $delay(s,t,q)$ (5.2).

The parameters of the mathematical function (s,t,q)correspond to the introduced component parameter *IoThreadRuntime*, *IoThreads* and *RequestQueues* to calculate the delay. These parameters are meant to be changed by the user. In case the parameter *IoThreads* is set to one, only the standard queue access delay will be issued because $m(t,q) = 0$. This configuration is useful in case of the *Sync_IOThread* component, where no blocked queues occur.

5.2.3.2 IoThread

In the asynchronous case, the user of the model can specify the I/O thread pool size by setting the *IoThreads* component parameter of the *Async_IoThread* component.

Component type	Parameter name	Description
Async_IoThread	IoThreads	Sets the size of the asynchronous I/O thread pool.
RequestGenerator	IoThreads	This parameter is used to calculate the blocking probability and queue access delay. Must be the same value as *Async_IoThread*'s IoThreads parameter.
	RequestQueues	Amount of request queues used. Required to calculate the blocking probability and delay of an I/O thread
	IOThreadRuntime	Runtime of the I/O thread. Used to calculate the delay for each failed queue access.
CapacityController	Capacity	Specifies the maximum capacity the *CapacityController* controls
	TimeTillReset	After this time (in μs), an acquired passive resource is released.

Table 5.2: Component parameters

This parameter determines the maximum amount or capacity of the passive resource which in turn reflects the thread pool size. By changing this parameter, the user can influence the number of I/O threads asynchronously working on request queues. The corresponding component parameter of *RequestGenerator* must be adjusted, as well, to calculate the proper blocking probability. In the synchronous case, the user cannot limit the amount of I/O threads.

In contrast to the amount of I/O threads, the user should have no influence on the request processing time and therefore the CPU resource demand the I/O thread issues in its `handleRequest` RD-SEFF, neither in the synchronous nor in the asynchronous case. This parameter is configurable and determined in chapter 6.

5.2.3.3 CapacityController

The *CapacityController* component was introduced to model the throughput constraint of a *channel* which connects the IBM system with the storage hardware. The following explains the parametrization of this component using the example throughput constraint of such a channel. The channel has a throughput of 65000 requests per second. Because the simulation engine cannot process so many requests per second and because all delay resource demands in this model are specified in μs, the parameter of the *CapacityController* must be scaled to $\frac{650 Requests}{10^4 \mu s}$. Therefore, the capacity of the *CapacityController*'s passive resource is set to 650 and the DELAY resource demand of the *InternalAction* `delay` to $10^4 \mu s$. To ease the change of these parameters, they were defined as component parameters and their default values are set to the example values.

5.2.3.4 IoInterface and StorageHardware

Both the CPU and DELAY resource demands of the *IoInterface* and *StorageHardware* are configurable parameters. As they should not be changed, these parameters

were not defined as component parameters. Their values must be determined by measurements and calibrated to match the real system behavior. The calibration and fine-tuning of the corresponding resource demands of the `processRequest` and `execute` RD-SEFF will be explained in chapter 6.

5.2.3.5 Completion thread blocking

Because of the reasons explained in section 5.1.2, it is not possible to model the queue locking without enormous effort and too hard to make proper assumptions about the blocking probability.

Therefore, the decision made was to calculate the delay with the expression given in (5.3) with a constant blocking probability for each completion thread of $p = 0.1$. The thread runtime is assumed to be $r = 10\mu s$ and its standard queue access delay is assumed to be 10% of its runtime, $s = 0.1 \cdot 10\mu s$. Therefore, the `signalCompletion` RD-SEFF DELAY resource demand is `(0.1 * 10) * Binom(1,0.1) * Pois(0.1)`. The binomial distribution function with a probability of $p = 0.1$ ensures that 10% of all calls evaluate to 1. Then the value of `Pois(0.1)` defines, how often the thread was blocked with a Poisson distribution with a rate of 10. This value is then mutliplied by 10% of the completion thread runtime.

As the thread runtime is $10\mu s$, it demands the CPU for this time. The amount of CPU cycles required during this time can be calculated by $c = r \cdot \text{CpuProcRate} = 10\mu s \cdot 1.7 \cdot 10^9\text{Hz} = 1.7 \cdot 10^4$. This is the value of the `signalCompletion` RD-SEFF CPU resource demand.

5.3 Summary

This chapter described the implementation of the architecture proposed in chapter 4 as a model instance of PCM. This was not a straight-forward process, because of the PCM-foreign domain of the modeled system. Hence, the modeling of some components was not easy, e.g. the modeling of the throughput constraint. Although these facts complicated the modeling process, this work could show that it is generally possible to use PCM in different domains. However, some assumptions and abstractions had to be made to achieve this target. The following chapter will validate if the model behavior matches the system behavior.

6. Synchronous model calibration and validation

This chapter explains the calibration and validation of the synchronous model implemented in the previous chapter. The evaluation of the asynchronous model will be discussed in detail in the following chapter 7. In this approach, calibration means to measure the configurable model parameters and use these values to derive the proper resource demands to fit the model to the behavior of an existing prototypical system.

Creating an accurate performance model requires to understand the internal behavior of the system. Hence, the configuration approach follows the systematic approach of *Experiment-based Derivation of Software Performance-Models* by Happe [Hap08] to identify the performance relevant influences of the real system. The attained results improve the knowledge and understanding of the system's behavior which in turn supports the configuration of the performance model skeleton.

First, this chapter motivates and explains the experiments conducted to determine the main influence factors on the system's performance. The next section describes the test system and the attainment of further specific measurement data. Moreover, it explains which difficulties complicated the measurements and their interpretation. Then, the derivation of the model resource demands from measurements so they can be used in the PCM model instance is explained. After the calibration's description, this chapter discusses the simulation results of the parameterized synchronous model. Thereby, the focus is on the validity of simulation results compared to the measured values and the behavior of the real system. The final section summarizes the model calibration and discusses the main important observations made during the configuration process.

6.1 Experiments - Overview

At the beginning of this work the influence and impact of the different performance relevant parameters on the system's performance were unclear. The experiment results of the following experiments shall help to draw conclusions about the influence

of the variable performance relevant parameters. The variable parameters of the real system are the request size and type and the amount of CPUs assigned to the VL.

Based on the experiment-based derivation method by Happe (see section 2.6), the following GQM plan poses the main goal of the experiments. The explicit questions the experiments must answer will be described later in this section. Furthermore, hypotheses regarding the influences of parameters on the system performance will be stated. These hypotheses are based on the current knowledge of the system and shall be evaluated by the experiments. If not disproved, these hypotheses will be used to define the behavior of the test system the synchronous performance model must follow.

6.1.1 The Goal

According to the scenario-based GQM methodology used by the experiment-based derivation explained in section 2.6, one needs to specify a goal for the performance evaluation. The goal of this approach is the following:

Goal:	*Purpose*	Identify
	Issue	the performance influence
	Object	of request size, request type and amount of CPUs
	Viewpoint	from the user's point of view.

The goal focuses on the evaluation of the performance influences which can be observed externally, in particular the throughput of the system. It is assumed that this throughput mainly depends on three performance relevant factors, the request size, the request type and the amount of CPUs the VL is equipped with. The metric throughput will be explained later in this section.

6.1.2 Motivation of the questions

To achieve the main goal stated in the previous section, the following poses questions targeted at the influence of request size and type on the system's throughput. One further question shall answer the performance influence of the amount of CPUs on the throughput.

Request size and type As mentioned in previous sections, it is intuitive that the different properties of a request influences the performance of the system (e.g. the throughput). The size of a request has an influence on the performance because the VL must process and transfer the request from the user application to the I/O interface. For example, the bigger the request the longer it should take the storage hardware to execute it.

The request type influences the system's performance because READs and WRITEs can cause different response times when executed by the storage hardware or by the I/O interface. Nevertheless, it is unknown to what extend the system's performance is affected by these parameters. The question RequestType and RequestSize summarized in table 6.1 address these influences and shall improve the understanding of these performance factors.

Performance influence of request size, request type and amount of CPU			
	RequestSize	ResponseTime	CPU
Question	To what extend does the request size influence the system's throughput?	How does the request type influence the throughput of the system?	What influence on the throughput has the amount of CPUs?
Scenarios	READ requests WRITE requests	4KB 16KB 64KB 256KB	1CPUs 2CPUs
Metric	Throughput (Requests/second)		
Hypothesis	The throughput decreases linear to the request size	The throughput for WRITE requests is lower than for READ requests	With more CPUs, the maximum throughput increases

Table 6.1: Questions and hypotheses concerning the performance relevant factors

Amount of CPU Another factor influencing the performance of the VL is the amount and the processing rate of CPUs attached to the VL's partition. The more power is available, the more and the bigger requests can be processed per unit of time. As the CPU power grows, other factors can limit the performance, e.g. the channel capacity or the storage hardware delay. Question CPU (table 6.1) addresses the influence of a different amount of CPUs assigned to the VL.

6.1.3 Experiment design

The following experiments are conducted on a System z. This IBM system is a complex computer system, consisting of several layers including software as well as hardware. Hence, the experiment setup was designed to get an approximation of the system behavior for a typical operating system and I/O hardware configuration. All of the experiment measurements were made by the IBM Germany Research & Development GmbH, because the test system was a shared resource with restricted access.

IBM uses a proprietary tool to measure the system throughput. Basically, this tool is a workload generator and is able to vary the request size and type issued to the system while measuring the system's performance. Furthermore, it provides the possibility to vary the amount of requests issued to the system. Such a process can be compared to an artificial user or an application which generates requests, called *request producer* in the remainder. Thereby it can be observed how fast (with how many users) the maximum throughput is reached. To test the throughput under a varying amount of CPUs, this workload generation tool must be re-run with a different VL CPU configuration.

All experiments and measurements in the following are executed on a System z9 with 48 processors and 128 Gigabyte of memory. The storage controller is a DS8000 and connected via 4Gbit/s FCP (Fiber Channel Protocol) channels. To avoid the bandwidth limit, the experiment setup was configured with 4 channels.

The experiments itself are designed as follows. Each experiment shall identify the influence of one performance relevant factor. Hence, the independent variables in the following experiments are request size, request type and amount of CPU. Furthermore, for each experiment the amount of request producers is increased in equal steps to observe the system under different conditions (little load up to high load).

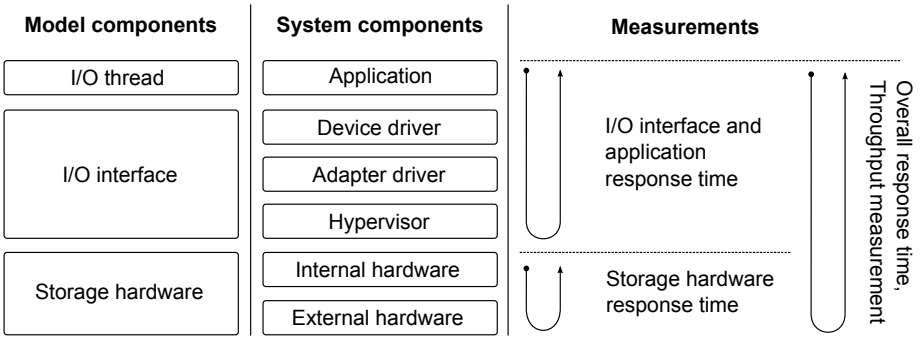


Figure 6.1: Measurement setup architecture

An example configuration of such an experiment is the following. Assuming that the influence of the request size shall be measured, there are different measurement runs for different request sizes, 4, 16, 64 and 256 kilobytes (KB). All other parameters are fixed during these runs, e.g. 1 CPU, only READ requests. Furthermore, to observe the performance impact under different load conditions, the workload generated increases by increasing the amount of request producers (1, 2, 4, 8 ... 256).

The performance metric measured in the experiments is always the throughput. 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 IBM proprietary tool measures the throughput as follows. At first, the amount of requests (R) can be measured by counting the amount of completed requests which returned to the workload generating application. The period of time (T) in which the requests are counted is the execution time of the workload generator. Then, the throughput can be calculated as the ratio of requests per time.

However, to calibrate and parameterize the resource demands of the model components, response time measurements must be conducted. The following section explains both, the throughput and response time measurements, conducted by IBM.

I/O stack

To understand how throughput and response time measurements were taken, this section explains a conceivable I/O request handling architecture for a typical operating system and I/O hardware (see figure 6.1). For these experiments, Linux was chosen as operating system, because it provides measurement tools, such as the FCP statistics tool [dev09] to obtain more detailed hardware results . Basically, the I/O stack can be divided into the sections depicted in figure 6.1.

On the top, there is the request issuing application (e.g. the workload generator), followed by the device driver and adapter driver. The next layer is the zHypervisor which is on top of the hardware. Compared to the model, the I/O thread corresponds to the application (the VL), the device driver, adapter driver, and hypervisor are represented by the I/O interface and the internal and external hardware by the storage hardware component.

The different sections of this architecture and response times and throughput were measured with different tools. The overall throughput and response time is the amount of requests issued by the application and was measured with an IBM proprietary tool. To measure the I/O interface response time and storage hardware response time, the FCP statistics tool was used.

For this work, the different layers were summarized to two main components corresponding to the modeled components. The layers can be classified into storage hardware response time (the time spent outside the machine) and I/O interface response time (which is the time spent in the device driver, adapter driver and hypervisor) (see figure 6.1). However, the FCP statistics tool does not measure the time spent in the application, device driver and adapter driver. Hence, it is not possible to measure the complete time spent in the OS, directly. It must be calculated by subtracting the storage hardware response time from the measured overall response time. Moreover, if one assumes that a constant amount of time is spent within the application, one can calculate the response time for the I/O interface component.

6.2 Experiment results - answering the questions

Based on the method proposed by Happe, the following defines scenarios and hypotheses for each question raised in the previous. The results of these experiment scenarios shall confirm the stated hypotheses about the performance influences, otherwise the hypotheses must be revised. Furthermore, these experiments shall increase the understanding of the system's behavior.

All of the experiment results relate to the throughput measured with the IBM proprietary workload generator tool for several specific amount of request producers. Hence, it is not a continuous measurement. The illustrations in the following depict the throughput measured for a specific amount of workload, interconnected by a straight line. Furthermore, the x-axis has logarithmic scale to clearly display all measurement points.

6.2.1 Question RequestSize

This question evaluates the influence of the request size (question RequestSize, table 6.1) on the system's throughput.

Scenario: In these scenarios, the varied parameter is the request size and takes the values 4KB, 16KB, 64KB and 256KB. In the READ scenario, the request type is set to READ whereas the request type is WRITE in the second scenario. In both cases there is only one CPU assigned to the VL. The measured results of this configuration shall allow conclusions about the influence of the request size on the overall throughput in case of READ and WRITE requests. Furthermore, varying the amount of request producers shall demonstrate the influence of the load on the system's throughput. The only metric in these scenarios is the throughput.

Hypothesis: For the described scenario, the hypothesis expects a decrease of the overall throughput which is linearly correlated to the size of the issued request. This assumption stems from the fact that the bigger the requests are, the longer their processing by the VL should take. For example, if a request must be copied during its processing, then the bigger the request is, the longer is the response time. This in turn influences the throughput. This linear dependency should be true for READ and WRITE requests. However, they might not show the same linear correlation.

Results: The results of the throughput measurements are collected and listed in table A.1. An illustration of the measurements is given in figure 6.3. For all request

types and size combinations, it shows an increase of the throughput as the amount of request producers is increased. This indicates that the system is not loaded with one request producer and at its limit with 256 request producers. Moreover, the throughput for a WRITE request is always below the throughput of the READ with corresponding size which indicates that WRITE requests must be more resource intense. Furthermore, as stated in the hypothesis, the throughput decreases as the request size increases. However, with in figure 6.3 it is hard to conclude any correlation between the request size and the throughput and thus is analyzed separately.

Though, for this analysis one has to consider the amount of request producers. In case of an increasing amount of users and hence an increasing system load, the correlation of request size and throughput might be distorted, because e.g. contention and/or concurrency effects might influence the throughput. Hence, to conclude the influence of the request size on the throughput, only the measured values for one request producer are taken into consideration. Figure 6.2 shows the correlation of throughput and request size. Apparently, the linear dependency is hard to conclude and hence, the hypothesis must be revised. A logarithmic correlation is much more fitting and can be concluded by using the measurements as input for the model ($f(x) = a * log(x) + b$)) in R [Dal08]. The resulting coefficients are $a = -1385.9, \quad b = 8244.5$ for READ and $a = -684.6, \quad b = 4710$ for WRITE requests. In case of READ, the correlation coefficient R^2 is 0.9892 and 0.9877 for WRITE.

In short, this experiment shows that the hypothesis is valid concerning the assumption that there is a correlation between request size and throughput. However, it is not linear but logarithmic. Furthermore, the measurements showed that the correlation is different for READ and WRITE request. The reason for this observation seems obvious if looking at figure 6.2 and 6.3. The bigger the request gets, the lower the maximum throughput. This is intuitive as the system is bounded by its resources the throughput converges to as the request size increases. Another conclusion is, that the throughput for WRITE requests decreases earlier and quicker than for READs. This indicates that WRITE processing must be more resource intense.

6.2.2 Question RequestType

This section discusses the influence of the request type on the system's throughput (question RequestType, table 6.1).

Scenario: In this case, the point of view onto the data depicted in figure 6.3 is different. Here, the request type issued by the workload generator varies whereas the request size is considered to be fixed. For example, one scenario compares the throughput results of a 4KB READ request with the one of a 4KB WRITE request. Further scenarios with similar settings shall examine the same question, but for different request sizes (16KB, 64KB and 256KB). As in the scenario of question RS, the amount of request producers is varied to observe the saturation of the throughput.

Hypothesis: For the four proposed scenarios, the hypothesis expects a throughput of READ request higher than a throughput of WRITE requests. This expectation is based on the assumption that WRITE requests require additional CPU resources within the OS. Moreover, it is assumed that WRITE requests cause more delay

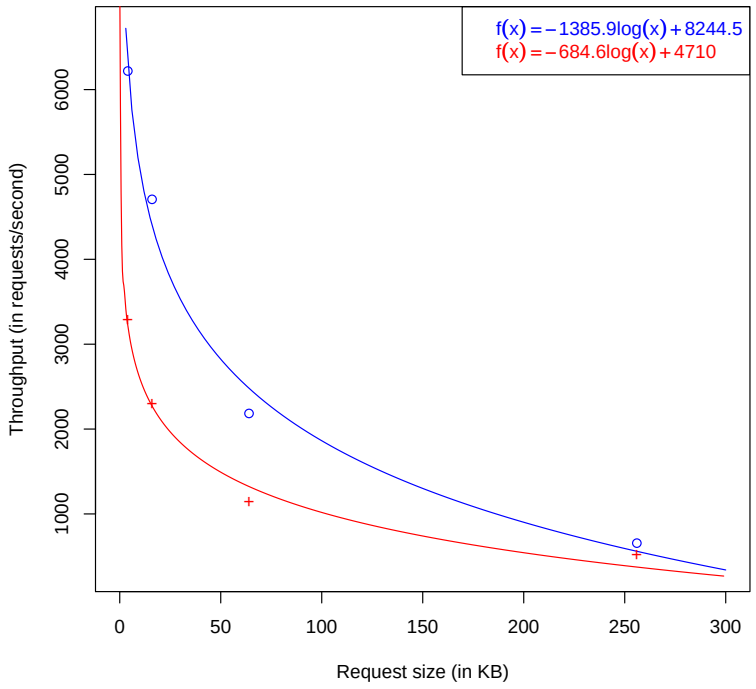


Figure 6.2: Logarithmic function fitted for READ (o, blue) and WRITE (+, red) requests

within the storage hardware then READ requests. This can influence the throughput under little load.

Results: Figure 6.3 illustrates the measured values for all scenarios, 4KB, 16KB, 64K and 256KB READ and WRITE requests. Obviously, the request type has an influence on the throughput. The throughput of WRITE requests is always below that of READ requests which confirms the assumption that WRITE requests are costlier than READ requests. However, as the request size increases, the influence of the request type on the throughput abates. This can be explained by the thought that the overhead for WRITES diminishes in comparison to the effort caused by the request size.

6.2.3 Question CPU

This question is targeted at the influence of the amount of CPUs (question CPU, table 6.1). The following scenarios and hypotheses shall reveal these influences.

Scenarios: To observe the CPU influence, two scenarios were designed. The difference is that in one scenario the reference system's VL is configured with one CPU. The other scenario uses two CPUs. In both cases, the request size is fixed to 4KB, 16KB, 64KB and 256KB and the request type is READ. As in the previous scenarios, the amount of request producers is increased.

Hypothesis: Assigning additional CPUs to the VL should increase the throughput because more requests can be handled at the same time, especially for bigger request sizes. Because one CPU should be sufficient to process the requests of little load

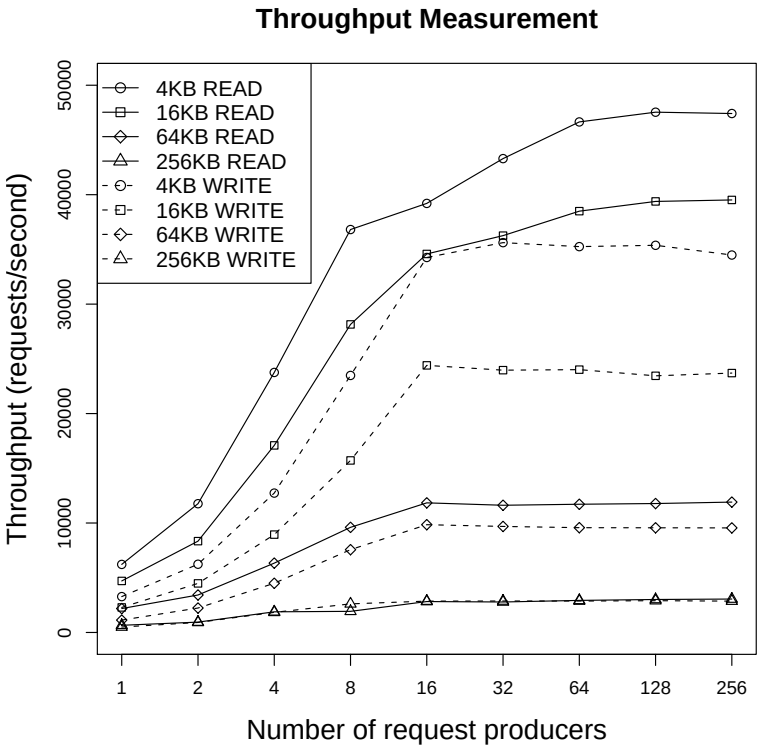


Figure 6.3: Throughput measurement for different request sizes and types with log-scaled x-axis

(few request producers), the benefit of two CPUs should be only observable for more than an initial amount of request producers.

Results: Figure 6.4 depicts the influence of additional CPUs. As one can see, further CPUs increase the throughput, but only if a significant amount of requests is generated by the workload generator. This demonstrates that the CPU becomes the bottleneck as the load increases. Additional CPUs slow down the emersion of this bottleneck. Furthermore, the throughput nearly doubles when adding an additional CPU for small requests, whereas this increase diminishes the bigger the requests get. On the initial throughput the additional CPU has only slight influences (see table A.2). This is because the response time for one request is not affected by the amount of CPUs, but would rather be affected by the CPU power. Hence, the benefit of additional CPUs is in scenarios with workloads of high load and high concurrency.

6.2.4 Discussion

The previous sections gave a feeling and understanding for the influences of the main performance factors. The hypothesis that the system's throughput linearly correlates with the request size was disproved and substituted by a logarithmic correlation. However, further measurements must be obtained to make better conclusions about the correlation of the request size and type and their influences within the operating system and the storage hardware.

Concerning the request type, the throughput of WRITE requests is lower than for READ requests, but only for small request sizes. The bigger the requests get, the less significant is the throughput difference of READ and WRITE requests of the

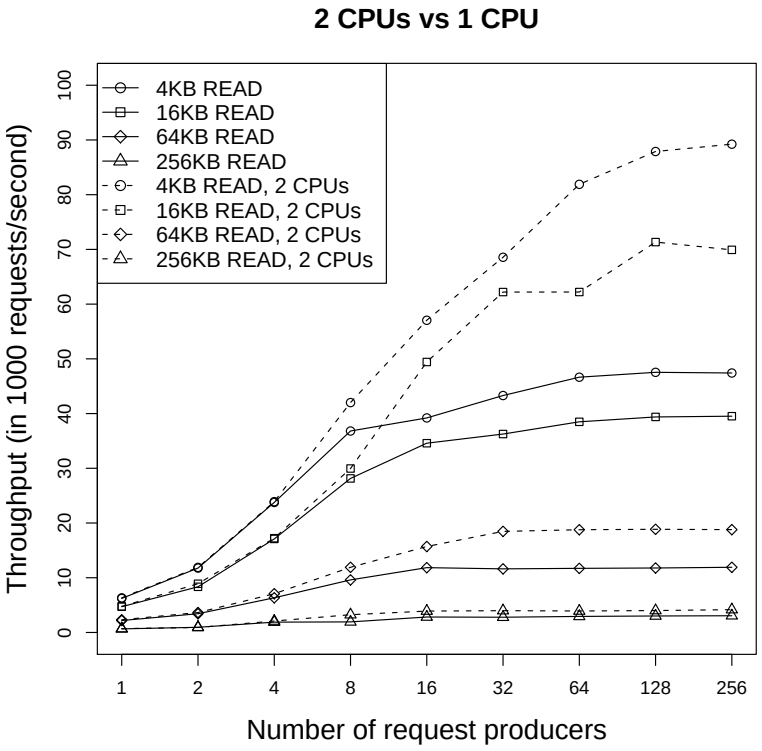


Figure 6.4: Measurements of READ requests with different CPU settings and log-scaled x-axis

same size. This effect can be explained by the fact that the overhead for WRITE requests abates in comparison to the overhead caused by the request's size.

Additional CPU power by assigning cores to the VL has no effect for few (< 4) request producers. Especially for big requests, the difference between the throughput of a VL with one or two CPU cores disappears.

6.3 Calibration of the performance model skeleton

This section explains the parameterization of the missing parts of the model skeleton proposed in chapter 5. Measurements of an existing System z will be used as a reference for the configurable model parameter values. The following sections explain these measurements for different parts of the system and the difficulties to obtain them. Furthermore, the validity and significance of these values is discussed and the conversion of the measured values to the necessary model input values will be explained. The whole model calibration process is guided by the results of the previous experiments. The target is to simulate the behavior of the real system as accurate as possible, while simultaneously using the measured values as unchanged as possible. However, approximations must be made to broaden the applicability of the model, e.g. to use other request sizes besides the ones which were measured.

6.3.1 I/O thread resource demands

The CPU runtime of the synchronous I/O thread and hence its CPU demand could not be determined exactly. Manual measurements at the real system targeted at

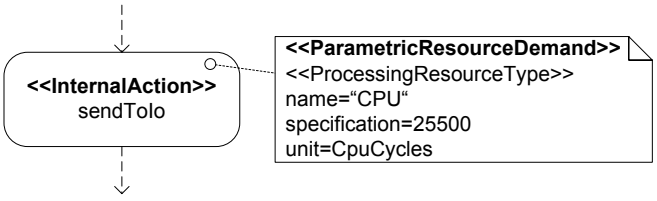


Figure 6.5: RD-SEFF of the *InternalAction* sendToIo of the I/O threads

determining the I/O thread runtime did not yield explicable and satisfiable results. The measured values were in the range of $10ms$ to $100ms$ which intuitively seems too high for the work the I/O thread has to do. Recall that the I/O thread basically copies pointers from the request queues to the I/O interface. However, with the results of a test series of an already existing, not fully configured model and the knowledge of the IBM experts, the CPU runtime of the synchronous I/O thread ($r_{IOThread}$) was determined to be $15\mu s$ on a 1.7 GHz CPU.

The CPU resource demands within RD-SEFFs must be specified in CPU cycles. The amount of CPU cycles of the I/O thread ($c_{IOThread}$) can be calculated by $c_{IOThread} = r_{IOThread} \cdot r_{CPU}$, where r_{CPU} is the processing rate of the CPU. The hereby calculated value of CPU cycles can be entered directly as the CPU resource demand of the *IoThread* component's RD-SEFF (see figure 6.5). As the runtime of the I/O thread on a 1.7 GHz CPU is set to $15\mu s$, the corresponding CPU cycles value entered in the *InternalAction* of `handleRequest` must be set to 25500. An example is given in figure 6.5 for the RD-SEFF `handleRequest` of the synchronous I/O thread.

6.3.2 I/O subsystem resource demands

The following section describes and discusses the detailed measurements of the I/O request handling stack (figure 6.1). These values are required for several reasons. First, they further improve the knowledge of the performance relevant parameters, especially for the different resource demands of READ and WRITE requests. Furthermore, these values help to configure the still not parameterized components *IoInterface* and *StorageHardware*.

6.3.2.1 Measurement results and interpretation

The measurements explained in the following were taken in the same hard- and software system already used for the experiments described in section 6.1.3. The measurements of the mean response time of *I/O interface + I/O thread*, the *storage hardware* and the *overall response time* were collected while the workload generator was issuing requests with one request producer. The restriction to only one request producer avoids the probability of measurement distortions caused by side effects like contention. The option to ignore caches was set and the data was collected for the same request type and size configurations as in the experiments. This was considered to be sufficient to determine the correlation of request size/type and response time.

The resulting data provided very detailed measurements of every layer of the I/O request handling stack (see figure 6.1). These values are reduced to the values listed in table 6.2. The *I/O interface + I/O thread* part of table 6.2 denotes the time the request spent in the OS and application part of the I/O subsystem. This time

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

Table 6.2: I/O Interface + I/O thread, storage hardware and overall response times and system throughput

is assumed to require CPU resources, only. In contrast, the *storage hardware* part is the response time of the storage hardware and is modeled by a DELAY resource which has no contention effects.

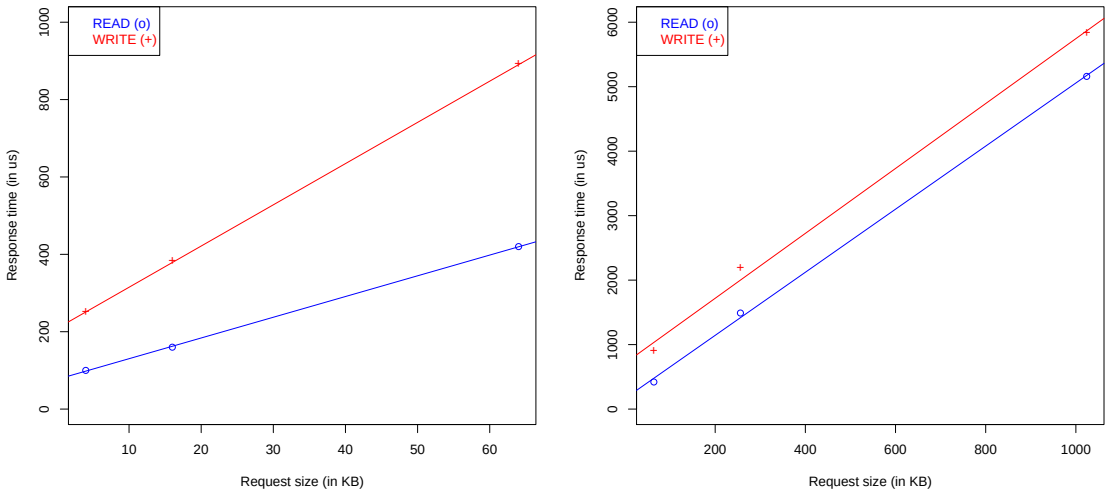
These detailed measurements help to recognize the correlation of request size, request type and the response times. However, they also bring up further questions because these values do net seem to fit to the throughput experiments. Each measured throughput is significantly below the throughput measured in the experiments. The only difference between this measurement setup and the one of the experiments is the additional FCP statistics tool. Hence, an explanation for the deviation could be that the additional FCP statistics tool slows down the request handling process. A comparison of the overall response time of the experiments with these measurements confirmed this assumption as it was clearly lower than the overall response times of table 6.2.

Another peculiar observation is that the response time of *I/O interface + I/O thread* for WRITE requests is below the ones for READ requests. This would imply less resource demands for WRITE requests. However, this conflicts with the previous experiments which showed that WRITE requests are more resource intense.

Thus, for the I/O interface part, further considerations must be made which will be discussed in section 6.3.2.3. As there are no contradictions concerning the storage hardware measurements, they are assumed to be valid and can be used directly, explained in the following section.

6.3.2.2 Calibrating the *StorageHardware* resource demands

Because the storage hardware resource is a DELAY resource whose resource demands are specified in μs, the measured values can be used as model input without any transformations or calculations. Based on the *storage hardware* values of table 6.2 it is possible to fit a linear model to the data set. The results are linear equations expressing the correlation of the request size and their corresponding response times of the storage hardware, depending on the request type. The good correlation coefficient R^2 of 0.9989 for READ and 0.9863 for WRITE requests indicates that the request size is linearly correlated to the throughput. However, the configuration showed that one can yield better simulation results if the data set is split into



(a) (b)

Figure 6.6: Linear correlation of request size and storage hardware response time for small (6.6a) and big (6.6b) requests

Request	Linear equation	R^2
READ	$4.95 \cdot reqSize + 115.31$	0,9992
WRITE	$5.35 \cdot reqSize + 446.84$	0,9897
READ, $\leq 32KB$	$5.36 \cdot reqSize + 76.67$	0,9998
READ, $> 32KB$	$4.89 \cdot reqSize + 165.00$	0,9993
WRITE, $\leq 32KB$	$10.65 \cdot reqSize + 208.34$	1
WRITE, $> 32KB$	$5.04 \cdot reqSize + 711.67$	0,9959

Table 6.3: Storage hardware response time linear equations and corresponding correlation coefficients. The parameter of the equation is the request size.

small and big requests. Figure 6.6 shows these linear correlations for small and big requests. The blue (red) lines indicate READ (WRITE) requests and 'o' ('+') the measured values. The resulting linear equations and their corresponding correlation coefficient are depicted in table 6.3.

To model the difference between READ and WRITE requests, the RD-SEFF of the *StorageHardware* component must distinguish between these request types. Furthermore, it must discern between small and big requests. This is modeled by a *GuardedBranchAction*. Then, the linear equations for each different request type and size are used to calculate the DELAY resource demand depending on the request size. This is modeled by entering the linear equation as StoEx into the DELAY resource demand of the corresponding *InternalAction* (see figure 6.7).

6.3.2.3 Calibrating the *IoInterface* resource demands

The CPU resource demand must be derived from the response times of the *I/O interface + I/O thread* depicted in table 6.2. However, comparing these response time measurements with the results of the experiments depicts contradictions, challeng-

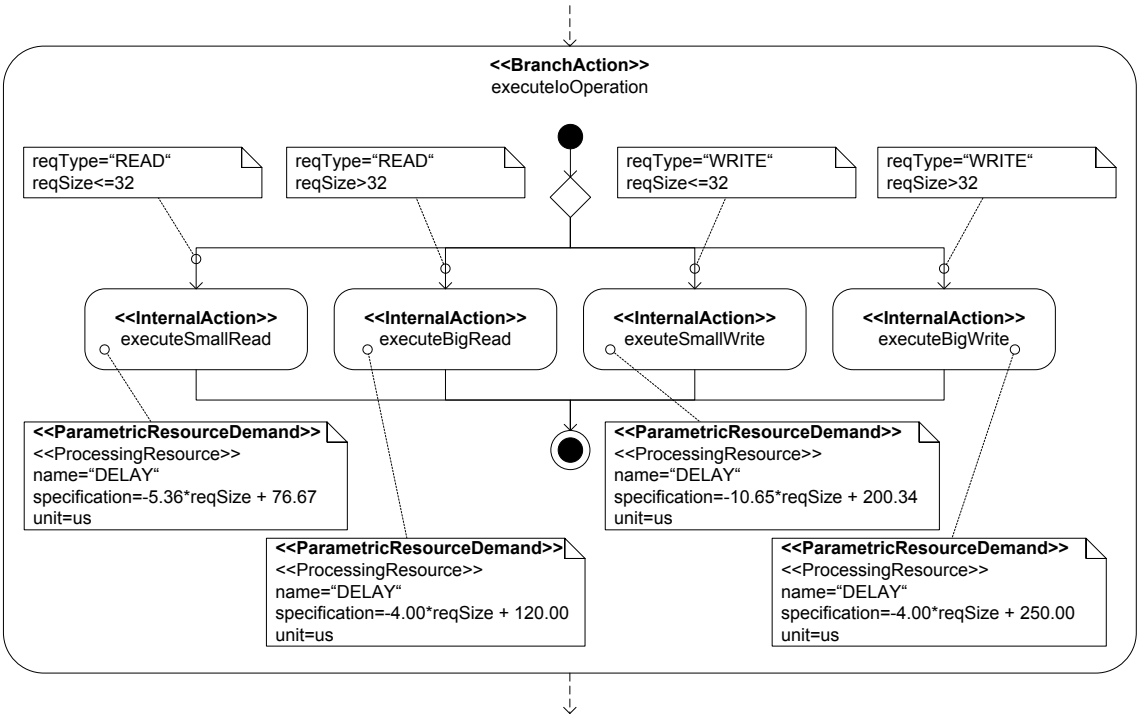
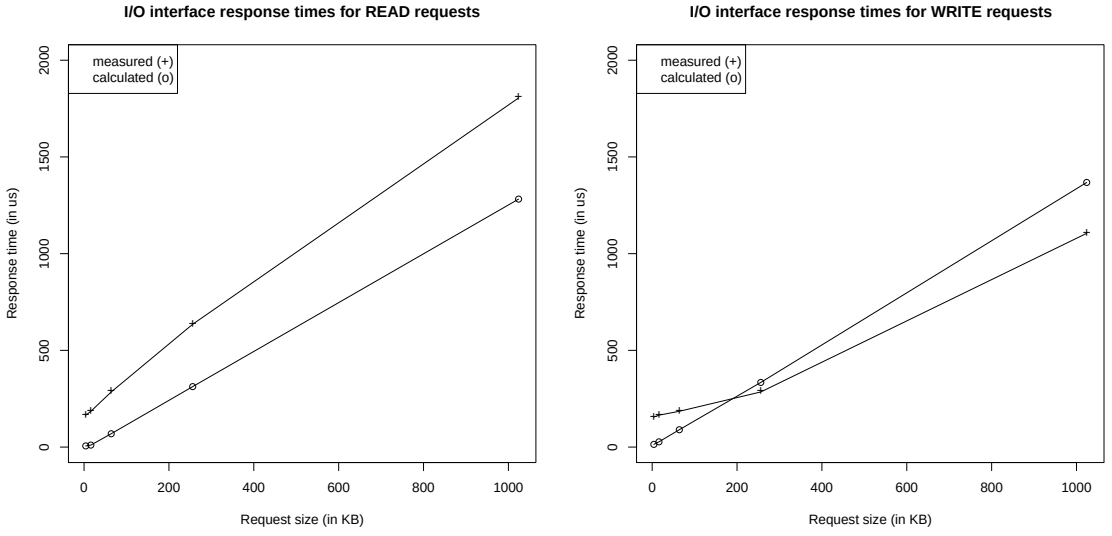


Figure 6.7: *BranchAction* executeIoOperation of the *StorageHardware* RD-SEFF

ing the validity and significance of the response time measurements. The following example calculation for 4KB READ requests demonstrates these incompatibilities.

The measured response time value of the I/O thread and the I/O interface is $180\mu s$ for 4KB READ requests. The reciprocal of $180\mu s$, approximately 5555 requests per second, is the theoretical amount of requests the model can process per second if the CPU is fully used. This value is clearly below the measured throughput of 47416 requests per second. Hence, using the measured values as CPU demand for the model components would lead to wrong simulation results.

In turn, one can calculate the maximum time the CPU is allow to require per request to achieve the measured maximum throughput. This is the reciprocal of the measured throughput. However, this is only valid if the CPU is the limiting factor. For 4KB READ requests, the maximum throughput measured was 47416 Requests per second which leaves a theoretical time of about $21.09\mu s$ per request. This theoretical value already includes scheduling and other overhead as well as the I/O thread runtime. When subtracting the I/O thread runtime of this calculated CPU runtime, the actual CPU demand of the I/O interface is approximately $6.09\mu s$. This is the value the I/O interface is allowed to require at most if the model shall reach the desired throughput measured in the experiments. The difference of measured and calculated value can be explained by delays caused by delays which occurred during the request processing and were captured by the measuring tools. Or the measurement tool caused additional overhead which delayed the request processing e.g. because of contention effects. Hence, the I/O interface component of the model needs to specify not only a CPU resource demand, but a DELAY resource demand, too. However, these computations are only feasible for a high load. Then, the storage hardware delay can be disregarded, because the CPU resource is assumed to be the bottleneck and thus the throughput is completely dependent of the CPU load.



(a) (b)

Figure 6.8: Measured (+) and calculated (o) I/O interface response time for READ (6.8a) and WRITE (6.8b) requests without I/O thread

	CPU demand		DELAY demand
Request type	Equation	R^2	
READ (≤ 32)	$5.29 \cdot 1.04^{reqSize}$	0.9997	$-(0.25 \cdot reqSize) + 28$
READ (> 32)	$(1.26 \cdot reqSize) - 11.43$	1	0.0
WRITE	$(1.33 \cdot reqSize) + 3.85$	0.9999	0.0

Table 6.4: Equations to calculate CPU resource demands and the DELAY resource demand of the I/O interface in μs

If the target is to match the throughput measurements of the experiments, only a model configuration according to the previous considerations leads to reliable results. If the target is the lower throughput obtained while measuring the I/O subsystem, then it is possible to use the measured values, directly, and the measured throughput is simulated, too. Hence, one could think of repeating the experiments with the hardware measuring tool and hence measure the decreased throughput for each request type and size and for each amount of request producers to get compatible throughput and response time measurements. However, because of the complicated measurement setup to measure both throughput and response times, it is not possible to re-run the experiments with maintainable effort.

The previous calculation can be repeated for all other request type and size combinations and with the assumption that the I/O thread runtime is $15\mu s$. The results are graphically displayed in figure 6.8a for READs and in figure 6.8b for WRITEs respectively, compared to the measured CPU resource demand.

Furthermore, figure 6.8 shows the CPU demand characteristic calculated according to the maximum throughput. For small request sizes, the increase of the CPU resource demand is not as intense as for big requests. This indicates a possible constant standard overhead for each request to be processed. This overhead becomes insignificant in relation to the resource demand needed to process big requests. How-

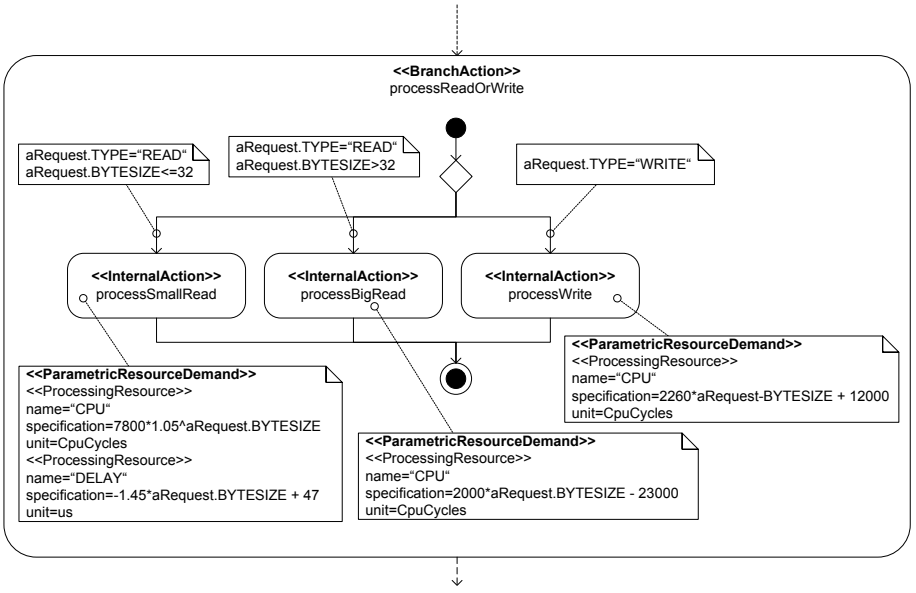


Figure 6.9: *BranchAction* processRequest of the RD-SEFF of the *IoInterface* component

ever, this behavior is observed for READ requests, only. Hence, the RD-SEFF of the I/O interface component has to distinguish between small and big READ requests and the WRITE requests. Splitting the READ CPU resource demand into small and big requests yields higher correlation coefficients. Especially for small READ requests, an exponential model shows a better correlation in comparison to a linear model ($R^2_{exp} = 0.9997, R^2_{lin} = 0.9832,$). For WRITE request, a distinction of small and big requests was insignificant. Moreover, an exponential regression for small WRITE requests showed a worse correlation than the linear regression ($R^2_{exp} = 0.7759, R^2_{lin} = 0.9999,$). The correlation of request size and I/O interface response times listed in table 6.4. As the calculated delays of the I/O interface did not show any specific and explainable correlation, these DELAY resource demands were manually adjusted and configured to best-fit the throughput measurements of the experiments. However, this delay is necessary for small READs to achieve the same initial throughput as measured in the experiments. It can be explained by periods the I/O interface was idle for some reasons, which was captured by the measurements, too.

The results of these considerations and calculations can be used to determine the CPU resource demand in CPU cycles. The CPU resource demand must be converted from time r to CPU cycles c. This can be achieved by calculating $c = r \cdot p$. The processing rate p of the CPU is known and amounts to 1.7 GHz. Then, the values of table 6.4 can be used to calculate and specify the RD-SEFF (figure 6.9) of the *IoInterface* component. A *GuardedBranchAction* checks the three different types of request size/type combinations. Then the proper resource demands can be specified as StoEx according to the determined equations to issue resource demands in the corresponding *InternalActions* (see figure 6.9).

6.3.3 Final calibration

The purpose of this final calibration is to calibrate the model with its CPU and DELAY resource demands in such a way that it matches the throughput measurements

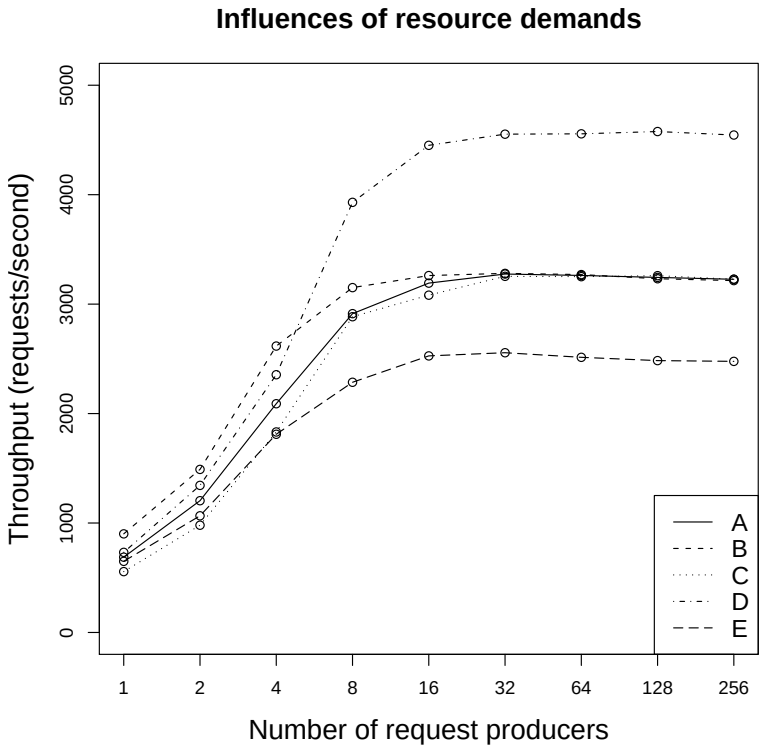


Figure 6.10: Influences of different resource demand settings. Curve A is the simulation result with the determined configuration setting for a 256KB READ request. Curve B and C are simulation result for 30% less and 30% more DELAY resource demands (storage hardware) respectively. Curve D and E describe the characteristic for 30% less and 30% more CPU resource demand (I/O interface).

with less then 10% relative error. The key values to reach during this calibrations are the i) initial throughput (the system is under litte load, only one request producer) and ii) maximum throughput (the system is under high load, 256 request producers).

During this final calibration process, it became evident that if one tunes the DELAY resource demands, the characteristics of the throughput curve stays the same, but the curve is moved left or right, with an higher or lower initial throughput. If the CPU resource demand is changed, the maximum throughput is decreased or increased, whereas the initial throughput is unaffected. These observations are graphically depicted for a 256KB READ request in figure 6.10, the measured simulation values are listed in table A.4. Hence, to fine-tune the initial (maximum) throughput, the resource demands of the storage hardware (I/O interface) must be adjusted. Furthermore, if for example the initial throughput is to low for all the big READ requests, one has to adjust the gradient of the linear equation.

With this considerations, it was possible to keep the relative error for almost every key value below the 10% margin (see table A.3), achieved by little adjustments to the correlations of request size and type determined in the previous sections 6.3.2.2 and 6.3.2.3. The resource demands finally used within the RD-SEFFs of the synchronous model are listed in table 6.5. The validity of this configured model will be discussed in the following.

	CPU (in CPU cycles)	DELAY (in μs)
I/O thread	25500	
I/O interface		
READ, ≤ 32	$7800 \cdot 1.05^{reqSize}$	$(-1.45 \cdot reqSize) + 47$
READ, > 32	$(2000 \cdot reqSize) - 23000$	
WRITE	$(2260 \cdot reqSize) + 12000$	
Storage hardware		
READ, ≤ 32		$(5.36 \cdot reqSize) + 76.67$
READ, > 32		$(4.00 \cdot reqSize) + 120.00$
WRITE, ≤ 32		$(10.65 \cdot reqSize) + 200.34$
WRITE, > 32		$(4.00 \cdot reqSize) + 450.00$

Table 6.5: Final CPU and DELAY resource demands after the calibration process

6.4 Synchronous performance model validation

For the validation of the synchronous performance model, another series of throughput measurements was collected. As in the previous experiments, the setup was completely the same (see section 6.1.3), only the type of requests was a mixture of 60% READ and 40% WRITE requests. A mixture of the request size was not possible because the workload generation tool did not support this feature. The throughput of this configuration was measured again for the four different request sizes (4KB, 16KB, 64KB, 256KB) and will be explained in the following section. After that, the influnece of additional CPU power will be discussed in the next section.

Before the measurements and simulations results are analyzed, hypotheses will be stated. These hypotheses must be confirmed by the measurement and simulation results. Basically, these hypotheses resemble the experiment's hypotheses. However, now they are targeted at the READ/WRITE mix measurement. Only the hypotheses concerning the influence of an additional CPU targets at the READ measurement of the experiment in section 6.2.3. The hypotheses are:

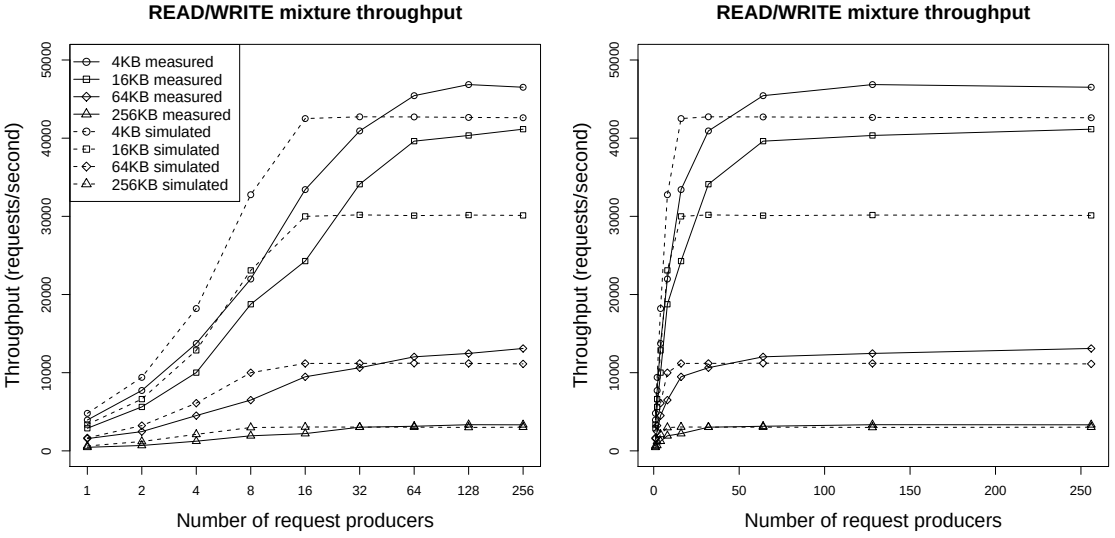
1. The bigger the requests of the mixture get, the lower is the throughput. The relation between the initial and maximum throughput of the different request sizes is the same as detected by the experiments.

2. Initial (maximum) throughput of simulation and measurement start (settle) at the same level. As for the experiments, quantitative errors are acceptable, but the qualitative prediction must not deviate.

3. If the CPU power is doubled, it is possible to process twice as many requests as with one CPU in case of the maximum throughput. Hence, the maximum throughput doubles. However, it does not influence the initial throughput, as the response time of one request is not influenced by an additional CPU.

6.4.1 READ/WRITE mixture

To evaluate the model, it was simulated and the simulation throughput results were collected for the same configurations as in the READ/WRITE mix through-



(a) logarithmic scale (b) linear scale

Figure 6.11: Comparison of measured and simulated throughput results for different READ/WRITE mixtures. Figure 6.11a is in a ld-scale and figure 6.11b has a linear scale.

put measurement (60% READs and 40% WRITEs for 4, 16, 64 and 256KB requests). To model the mixture of request types, the StoEx inside the *RequestGenerator* component which sets the request type has to be adjusted. The StoEx `IntPMF[(0;0.6)(1;0.4)]` arranges that 60% READ requests and 40% WRITE requests are generated. The simulation time was set to one simulation second and the amount of measurements at the `handleRequest` sensor was used as the amount of completed requests.

The measured and simulated throughput results of the READ/WRITE mix are depicted in figure 6.11. Both, the measurement and simulation confirm the first hypothesis with respect to the requested behavior, namely the throughput decreases the bigger the requests are. The second hypothesis is qualitatively confirmed, too, although the simulation results are deviating from the measured results. This qualitative comparison shows a good match of system and model behavior. However, the quantitative results show more deviation then expected. This will be discussed in the following.

In case of the initial throughput, the relative error of the mix is bigger than the errors of the uniform request type throughput measurements (see table A.5). However, a qualitative analysis of this deviation shows that the measured initial throughput of the mix is between the initial throughput of uniform READ and WRITE measurements (for 4KB, 16KB and 64KB). The simulation shows the same behavior since at least the same areas are hit.

However, the relative errors 8% (4KB), 27% (16KB), 15% (64KB) and 9% (256KB) show deviations in case of the maximum throughput. They can result from errors made by approximating CPU and DELAY resource demands with linear and exponential regressions. However, a test simulation run with the calculated values did

not lead to better results and thus demonstrated, that this is not the reason for the deviations.

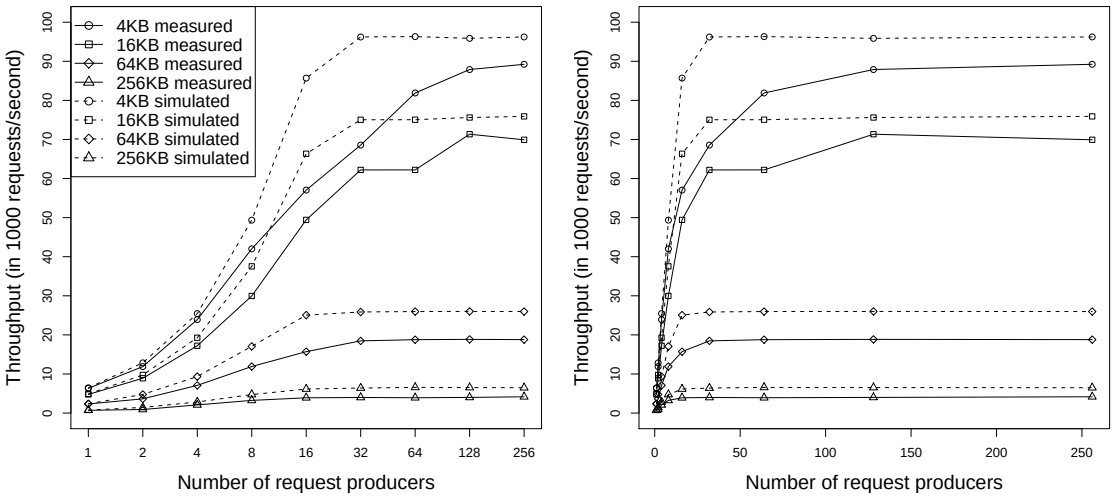
Another explanation for these errors can be derived from the throughput measurements. When focusing on the maximum throughput of READ and WRITE requests bigger than 16KB, it is peculiar that the throughput of the READ/WRITE mix is above the throughput of the READ requests. This is only explainable by the consideration that the WRITE requests require less CPU than the READs, what is confirmed by the separate I/O interface response time measurement but contradicted by the experiments.

However, as the model tries to achieve the throughput measurements of the experiments, the calculated CPU demand for WRITE requests was set to a higher value then for READs. Hence, with this model configuration in case of a 16KB request mix, the throughput of the model must decrease in comparison to the throughput of READs-only, as the READs are mixed with the more CPU-intense WRITEs. In contrast, in the real system the READs must be mixed with something less CPU-intense to explain the increase of the mix throughput above the READ throughput. Consequently, measured and simulated results drift apart, causing an even greater deviance.

Another explanation for the higher throughput in case of the measurement is that cache hits might have had an influence. Although the measuring application was configured to try to ignore caches, it is likely that cache hits occured, especially if there are two caches (one in the storage hardware and the other inside the VL's OS). Especially if there's a hit within the OS cache, the speedup could be influencing the throughput.

Furthermore, there are errors not quantifiable caused by measurements and calculations. Especially for small requests, a proper measurement of their very short response times is hard as other influences like timer resolution or even context switches can easily distort them. A correction of these errors, especially in case of the 16KB mix is not easy, as changing the model parameters would influence the results of the other simulation results.

To further minimize the simulation's errors, one would need more data to make better estimates about the resource demands. First, more data means more experiment runs to calculate mean response times for different request size/type configurations which could improve the quality for small request sizes. Second, a more discretized experiment design - which means more than four request sizes - could be helpful to make a better approximation of the request type and request size correlation. And third, a measurement setup which covers both throughput and I/O request handling component response times can lead to better suiting results of both throughput and I/O subsystem measurements. However, collecting this data causes tremendous effort, especially if caches must be completely avoided or the OS code must be instrumented to get more detailed results. Based on the measurements results and their accuracy, the simulation provides a good qualitative and quantitative prediction accuracy in case of synchronous request handling for READ or WRITE requests only. For a request mixture the results can be accepted under the exception, that the prediction is only qualitatively valid. However, the deviations are explainable but their resolving would require further analysis of the real system which might require an extension of the model.



(a) logarithmic scale (b) linear scale

Figure 6.12: Measured throughput for 2 CPUs and simulated throughput for doubled processing rate with ld-scale (6.12a) and linear scale (6.12b).

6.4.2 CPU power

To examine and analyze the influence of additional CPU cores, the idea was to integrate the exact scheduler created by Happe into PCM. The advantage of this exact scheduler is that it provides an exact simulation of scheduling policies which consider multicore CPUs. As the provisional integration of this exact scheduler was completed at a late stage of this work, a first analysis examines the CPU's influence on the throughput by a doubled CPU processing rate. In a second setup, the exact scheduler is used to basically test the influences of the scheduling. The following describes these analyses, starting with the doubled processing rate.

During the first comparison of measurement with 2 CPU cores and simulations with doubled processing rate, both results contradicted each other, as the model predicted a significantly higher throughput for request sizes over 16KB. A revision of the measurement revealed, that with two CPUs the bandwidth restricts the throughput. Hence, the measurement was repeated with more channels connecting the storage hardware with the system. Accordingly, the throughput controller's capacity must be raised to 130000 requests per second, too. Another possibility is to model the bandwidth restriction by limiting the throughput controller to the limit of the bandwidth. For example, if 64KB requests are issued and the bandwidth is 8 GBit per second, the throughput controller's capacity must be set to $\frac{8 \cdot 10^9}{64 \cdot 10^3 \cdot 8}$ requests. Simulation runs with this configuration then are bandwidth-limited, too. However, it is the theoretical limit which is reached in the simulation and not the real bandwidth restriction. Hence, the solution without bandwidth restriction and extra channels is considered to be more expressive.

With this configuration, measurement and simulation results (see figure 6.12) confirm the third hypothesis, as the throughput almost doubles in both cases. Furthermore, it confirms the expectation that increasing the CPU power unburdens the I/O interface and hence increases the throughput. However, the throughput increase of the simulation with additional CPUs is more significant the bigger the requests get.

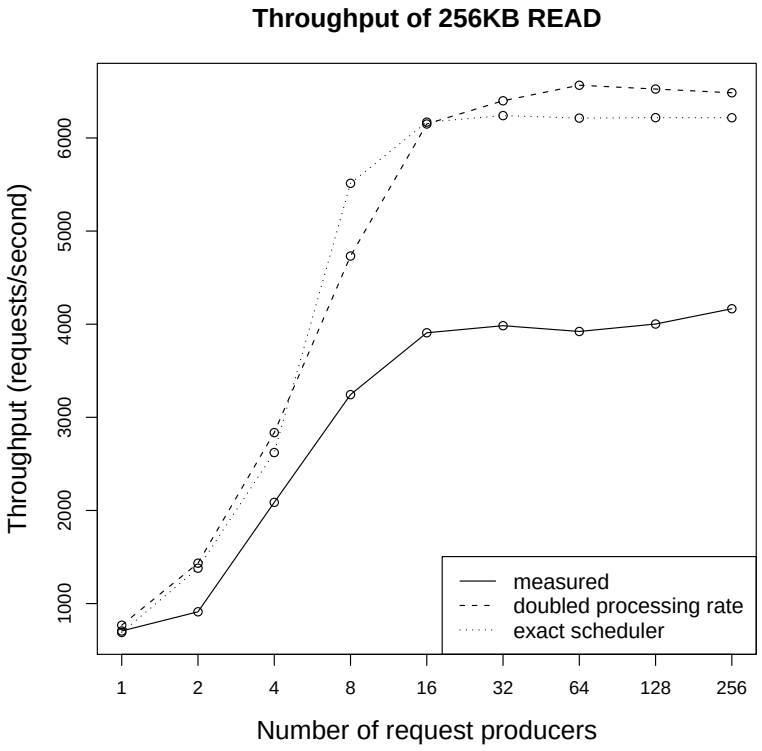


Figure 6.13: Throughput comparison of a 256KB READ request of the measurement results and the simulation results with a doubled processing rate and the exact scheduler

This is not observed in real system. An explanation can be the scheduling influences in the real system which are not captured by the model. Nevertheless, the simulation and measurement results with twice the CPU processing rate shows that the initial throughput is almost the same in both cases and only influences the maximum throughput (see table A.6).

For the exact scheduler by Happe, a special PCM workbench and PCM model had to be created. This PCM workbench provides the possibility to specify the amount of replicas of a resource container's processing resource and to set the processing resource's scheduling policy. Hence, to simulate the synchronous model with an additional CPU core the amount of replicas of the CPU was set to two. Furthermore, the exact scheduler's simulation time is based on milliseconds. Hence, each value referred to the simulation time like processing resources, resource demands, simulation time or the interarrival rate must be transformed to this unit.

The differences of a doubled processing rate and the exact scheduler are exemplary examined for a 256KB READ request and are graphically depicted in figure 6.13. As one can see, the maximum throughput of the exact scheduler is lower than for a doubled processing rate. This is observed for other request sizes, too (see table A.6). The explanation is probably the utilization of both CPU cores. In case of the exact scheduler, one CPU core is 10% idle. The reason for this idle time might be the fact that I/O threads or the I/O interface has to wait for the response of the storage hardware. However, it is remarkable, that the idle time is always approximately 10% for the second core for whatever request size or type. This is peculiar as the idle

time should be equal on both CPU cores. However, further simulations and alayses are required to better assess the influences of the exact scheduler, to understand its behavior and to check its correct integration.

Furthermore, it is remarkable that the exact scheduler's prediction results are better for both the initial and maximum throughput. Especially for the maximum throughput, the simulation results with 2 CPU cores are significantly lower than the results with doubled processing rate. This is explainable because the exact scheduler considers e.g. context switching which slows down the throughput. On the contrary, simply doubling the CPU processing rate does not consider such side effects and thus results in a higher throughput.

In short, the exact scheduler provides better prediction results than simply doubling the CPU processing rate. The analysis showed that adding additional CPUs to the VL improves the performance behavior of the system. However, scheduling effects must be taken into consideration. The exact scheduler can reflect these influences, but the measurements indicate more significant influences for big requests and heavy load.

7. Evaluation

After the calibration and validation of the synchronous model behavior in the previous chapter, this chapter evaluates the asynchronous model. This is performed in two ways. First, the performance relevant parameters are varied and the results are compared and discussed. And second, the synchronous and asynchronous design alternatives will be compared with respect to their performance. As there is no possibility to validate the results of the asynchronous model by means of measurable results, the plausibility and validity of the asynchronous simulation is discussed. The first section explains the simulation setup with open and closed workloads. The second section evaluates the variable model parameters and their influences on the performance as well as new identified cruxes of the model's performance behavior. The last section compares the simulation results of the synchronous and asynchronous model as far as possible to assess performance drawbacks and advantages of the design alternatives.

7.1 Asynchronous model setup

Unfortunately, there is no reference system or prototypical implementation one could measure the response times for the asynchronous components. However, the goal of this work was to construct a performance model to avoid the implementation of a performance prototype. Therefore, it is assumed that the resource demands of I/O interface and storage hardware do not differ from their synchronous counterpart. The same applies for the asynchronous I/O thread which has the similar runtime as the synchronous I/O thread. The only differences are the behavior of the asynchronous I/O thread, the asynchronous I/O interface and the possible delays because of blocked queue accesses as explained in chapter 4. To simulate the asynchronous model, the synchronous components must be replaced by their synchronous correspondents within the system diagram.

The model is tested under two types of workloads, open and closed. In a closed workload, one can specify the amount of users working on the system. In an open workload, the interarrival time specifies how much time elapses between the arrival

of two requests (see section 2.4). For the synchronous model validation and evaluation, the closed workload corresponded to the kind of workload generated by IBM's workload generation tool and hence, a closed workload was the best choice.

However, Schroeder et al. demonstrate that the type of workload issued to a system can have significant influences on the performance results [SWHB06]. Especially in the asynchronous model, an open workload is more reasonable because arriving requests should be independent of the processing of their previous request. Moreover, the authors show that as the amount of users of a closed workload increases, the closed workload approaches open ones [SWHB06]. The following examines the influences of both open and closed workloads.

The closed workload always has the same configuration as for the experiments in the previous chapter and the results are measured for a varying amount of request producers (1,2,4,...,256). The interarrival time of an open workload can be calculated according to $L = \lambda W$ (Little's Law [Lit61]), where L is the amount of requests in the system and W the amount of time a request spends within the system. Hence, for an open workload the interarrival time (or arrival rate λ) fully utilizing the asynchronous model is the reciprocal of the maximum throughput per second. If the arrival rate λ exceeds a system's processing rate p, the system gets overloaded.

7.2 Parameter influences

This section discusses the influence of variable performance parameters on the simulated system's performance, assessed by the throughput. The simulated system's performance behavior is observed and compared with different parameter settings as well as with closed and open workloads. If not stated otherwise, the following simulations are conducted exemplary with 256KB READ requests. All quantitative simulation results are listed in table A.7.

Amount of I/O threads

The first parameter setting observed is the VL's asynchronous thread pool size, thus the amount of asynchronous I/O threads. It was varied between the three values: 1, 10 and 100 threads. Concerning the simulation results, one can observe two different throughput metrics. The first is how many requests are issued by the I/O thread per unit of time. This value indicates the amount of requests currently processed by the system. It can be measured by the amount of calls returning from `handleRequest`. The other metric is the amount of requests which were actually completed. This is the amount of returned calls of `signalCompletion`.

The first peculiarity is that already one I/O thread reaches its maximum throughput for already one request producer. This maximum is roughly the same as the maximum throughput in the synchronous case. An additional amount of request producers has no influence on the throughput. However, this is only valid under the assumption that the request producer is able to generate requests in the same or a higher frequency as the asynchronous I/O thread can pass them to the I/O interface. But if the I/O thread's runtime is quicker than the rate in which requests are issued to the VL, one I/O thread is sufficient to handle the requests. This is independent of request size or type and is only influenced by the CPU load. Hence, the VL only

benefits from further threads if the requests arrive within a shorter period of time the I/O thread can process them.

Further noticeable is that the asynchronous I/O threads send more requests to the system than effectively completed. Hence, the system can be considered as "overloaded". For more than one I/O thread, if increasing the amount of request producers the throughput drops until the amount of request producers reaches the same amount as the number of I/O threads. Then, for a further increasing amount of request producers, the throughput is constant. This observation can be explained by the overload. At first, as long as the system is not used, the I/O threads can put requests into the system without being impeded. While the threads process the next requests, they are hindered by the previous requests currently processed by the I/O interface. Hence, the processing time and response time of the I/O thread increases. This delay continues to increase until a saturation is reached. This is the system's state when the I/O threads put the same amount of requests into the system as completed by the completion thread.

As the simulation results show, there is a difference between amount of requests in the system and actually completed requests. This difference is generated while the I/O thread(s) are issuing requests unimpeded by the I/O interface. However, the ratio of surplus requests and successfully handled requests converges to one the longer the simulation lasts. This indicates, that the simulation has a warm up phase until it reaches a steady-state.

In the following the influences of the amount of I/O threads with an open workload will be observed. The simulation results show that the maximum load of the model with the previous settings (256KB READ) is achieved for an interarrival time of $3.14 \cdot 10^{-4}$ seconds (1/3185 seconds per request). With this configuration, the CPU utilization for one I/O thread is 99,6% and the response time of the I/O thread is approximately $15.7\mu s$. A shorter interarrival time of the open workload leads to an overloaded system. This can be observed at the CPU utilization, where suddenly more than just two jobs are busy and in the I/O thread's response time which further increases.

Analyzing the influences of additional I/O threads, no effects on the system throughput or response time are observable. This behavior is explainable by comparing the response time of the I/O thread with the interarrival time. The interarrival time of $314\mu s$ is significantly higher than the response time of the I/O thread ($15\mu s$). Hence, one thread is sufficient to handle this load and thus the benefit of additional threads only observable if the thread's response time is higher than the interarrival time. Unfortunately this cannot be tested as the simulation breaks down because it is overloaded before the interarrival time can be sufficiently lowered.

However, a second simulation run with a more realistic interarrival time indicates an influence. This time, the interarrival rate is assumed to be exponentially distributed, with a mean arrival rate of approximately 3180 requests, the maximum throughout optimally utilizing the system. This exponentially distributed arrival rate assures, that the interarrival time is varied and the system does not settle in a steady-state. The cumulative distribution function shows (figure 7.1) the fluctuation of response times for `handleRequest` with 1, 10 and 100 asynchronous I/O threads. One can see, that the dispersion of the response times abates and the mean response time

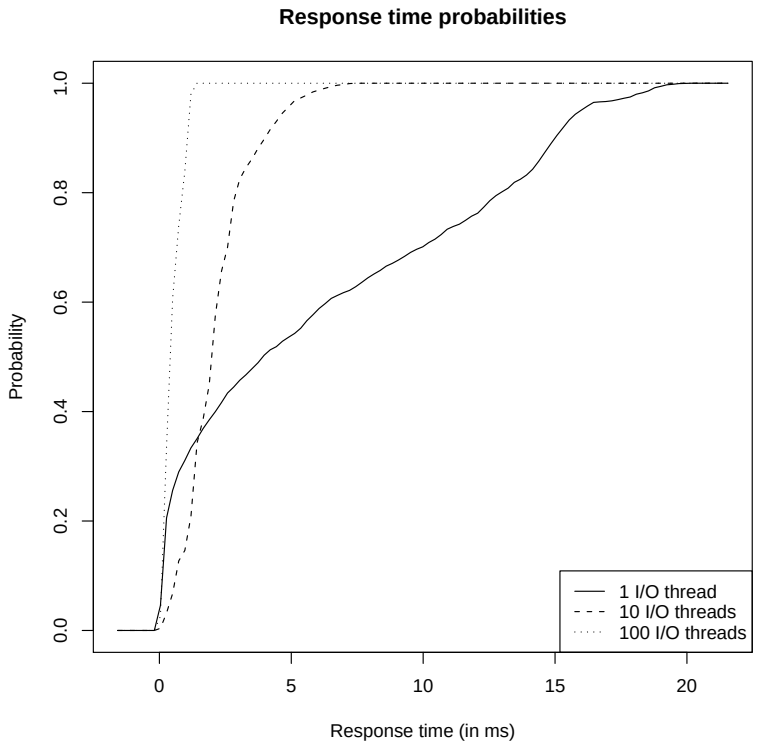


Figure 7.1: Cumulative distribution function of the response time for `handleRequest` of 1, 10 and 100 asynchronous I/O threads

shortens if the thread pool size increases. This indicates the advantage of thread pool in comparison to a single thread, namely the pool can better handle peak loads like the ones simulated by the exponential interarrival time.

Queue blocking

The following discusses the influences of queue blocking. Therefore, the previous results are used as reference results. In additional simulations the amount of queues was fixed to 256 for one test series and took the same value as amount of request producers in another series. The simulations shows that the delay of I/O threads or completion threads because of blocked queues has no significant influence, at least not if the system is fully loaded. The reason is that the blocking delay of about one to six microseconds has no significant consequences on overall response times of 80 milliseconds and more. However, it can have an influence under little load, as the overall response time is much shorter.

To test the influence of queue blocking in an open workload, the interarrival time was increased to $1.5ms$. This assures that no other influences besides queue blocking affect the I/O thread response time because previous requests are completely handled before a new request arrives. The first simulation run with one I/O thread and 256 request queues showed no blocking delays. The mean response time of `handleRequest` for 667 measurement points is $15.75\mu s$. The first observable difference occurs in a simulation run with 256 request queues and 100 I/O threads where blocked accesses can be registered in the response times. In total, the mean response time for 667 measurements is only slightly different ($15.83\mu s$). Even for 1000

threads and 256 queues, the throughput is not affected and only the asynchronous I/O thread mean response time is slightly increased to 16.49μs for 667 measurements. This demonstrates that the blocking delay is insignificant with respect to the overall system response time. However, in the worst case the response time of the I/O thread for one request might be approximately 17μs instead of 15μs. One could argue, that this difference can influence the overall response time for one single request because the request is sent to the I/O interface 2μs delayed. However, this delay is insignificant if the minimum overall response time for one request is approximately 1.45ms.

The same considerations apply for the completion thread. The difference of the system components response times because of the additional CPU resource demands of the completion thread or delays are marginal. They are insignificant for the mean response time or overall throughput of the system. Regarding one request, it has a slight effect on the overall response time (1447.3μs vs. 1477.4μs) and I/O thread response time (15.65μs vs. 15.75μs) if comparing synchronous and asynchronous model. However these influences disappear if comparing overall response time and throughput.

7.3 Synchronous and asynchronous model comparison

To compare the synchronous with the asynchronous model, both are simulated with an open workload with interarrival time of 3.14μs and 256 request queues. In the asynchronous case, only one I/O thread handles the arriving requests. However, a straight-forward comparison of both variants is not simple as the following shows.

At first glance, both simulation results seem similar (see table 7.1). In the synchronous case, the throughput is 3181 requests per second and in the asynchronous case, the completion thread signaled 3181 requests per second, too. Also the response times are approximately the same, as the mean response time per request in the synchronous case is 1.45ms and 1.48ms in the asynchronous case. The difference between the two approaches is that the CPU utilization in the synchronous case (96.4%, at most one busy job) is a bit lower than in the asynchronous case (99.6%, at most two busy jobs). This difference in utilization is almost exactly the overead caused by the completion thread which can be reconstructed by calculations.

Comparing the synchronous model with open and closed workloads assesses the influences of the workload on the system's throughput. The comparison reveals a slightly worse throughput of only 3121 requests per second in the closed workload case. The difference is even more significant when comparing the response times. The response time of one request for the open workload is 1.45ms whereas it takes 77.67ms in a closed workload. This response time is so much longer as the closed workload dramatically overloads the system. This overload is caused by the high concurrency of request arrivals generated by 256 request producers. Moreover, high concurrency on the CPU extends the response times of the CPU demanding components. Because the synchronous thread has to wait for the return of the CPU demanding I/O interface, the overall response time increases.

This indicates that the high load generated with the closed workload is not really appropriate in the asynchronous case. However, it is not possible to simulate the

Simulation Configuration	Response time [us] I/O Thread	I/O If + HW	Comp. Thrd	Overall	Throughput (requests/second) of handleRequest	signalCompletion
A-OW314-1	15.75	1441.63	20.01	1477.39	3185	3181
A-OW150-1	15.75	1431.63	10.01	1457.39	667	666
A-OW150-10	15.75	1441.63	10.01	1467.39	667	666
A-OW150-100	15.83	1431.65	10.01	1457.49	667	666
A-OW150-1000	16.49	1431.65	10.01	1458.15	667	666
S-OW314	15.65	1431.65	-	1447.30	3181	-
S-CW256	3694.00	73985.00	-	77679.00	3121	-

A-OW314-1	Async. Model, OpenWorkload, Interarrival Time 3.14*10^-4s, 1 I/O Threads, 256 Queues
A-OW150-1	Async. Model, OpenWorkload, Interarrival Time 1.5*10^-3s, 1 I/O Thread, 256 Queues
A-OW150-10	Async. Model, OpenWorkload, Interarrival Time 1.5*10^-3s, 10 I/O Threads, 256 Queues
A-OW150-100	Async. Model, OpenWorkload, Interarrival Time 1.5*10^-3s, 100 I/O Threads, 256 Queues
A-OW150-1000	Async. Model, OpenWorkload, Interarrival Time 1.5*10^-3s, 1000 I/O Threads, 256 Queues
S-OW314	Sync. Model, OpenWorkload, Interarrival Time 3.14*10^-4s
S-CW256	Sync. Model, ClosedWorkload, 256 Users

Table 7.1: Throughput and response time measurements for different simulation settings.

model with an open workload which generates a comparable load as the simulation would crash.

The main difference between synchronous and asynchronous model under an open workload with an exponentially distributed interarrival time is depicted in figure 7.2. It shows that the dispersion of the response time in case of the synchronous I/O thread is very high, comparable to that for one asynchronous I/O thread. Furthermore, the mean response time in the asynchronous case with several I/O threads is shorter and shows less dispersion as the synchronous I/O thread. However, a direct comparison of the response times is not possible as the response time in the synchronous case also comprises the response time of storage hardware and I/O interface. This is exhibited by the right-shifted cumulative distribution function of the synchronous model. The gap indicates the response time of I/O interface and storage hardware.

7.4 Discussion

Concerning the performance of synchronous and asynchronous request handling, there are some observable slight differences. However, they strongly depend on the kind of load exposed to the system. Assuming that a closed workload with a high amount of users in the synchronous and an open workload in the asynchronous case simulate an equal load, the synchronous version has slightly less throughput. In short, one can conclude that the differences of both approaches have only marginal influence on the throughput because their similarities (I/O interface and storage hardware) dominate the performance behavior of the system.

Hence, the decision between synchronous and asynchronous I/O can be made based on the general properties of synchronous and asynchronous I/O. One advantage of synchronous I/O is that it is easier to implement and better to maintain. For example, if a request is not returned for any reason, it is easier to determine which thread is affected. Furthermore, it has a intrinsic mechanism preventing the system from overloading as a new request can only be processed if a previous one is completed. However, synchronous I/O has the disadvantage that the VL's thread pool must

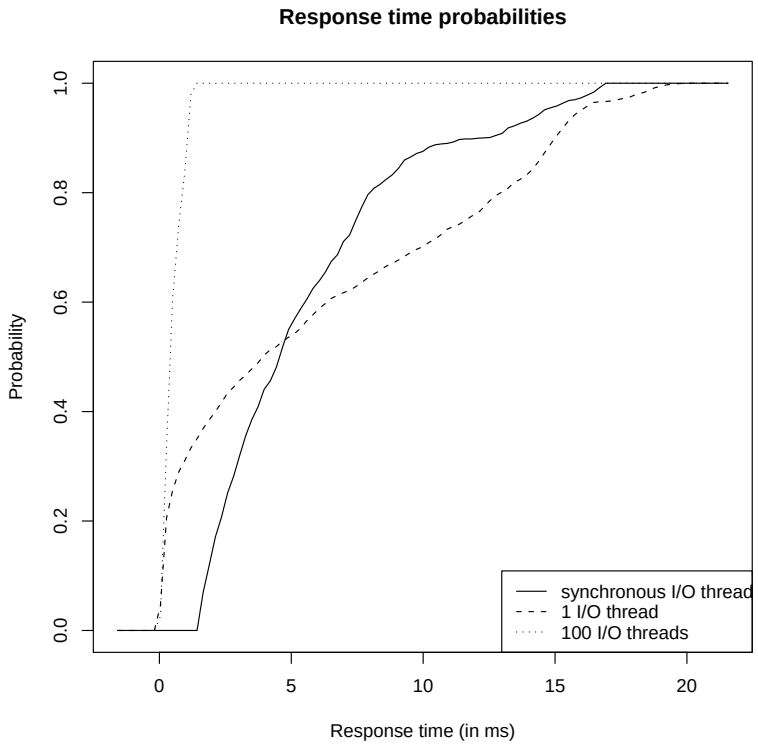


Figure 7.2: Cumulative distribution function of the response time of `handleRequest` for 1 and 100 asynchronous I/O threads and the synchronous I/O threads

provide enough threads, whereas in the asynchronous case already ten threads seem to be sufficient according to the results of this work. Furthermore, asynchronous I/O is more flexible and can better handle peak loads as the simulations with an exponentially distributed interarrival time demonstrated. However, asynchronous I/O has the disadvantage, that it is likely to overload the system if no preventing mechanism are implemented.

Furthermore, the evaluation of the models shows that the influences of the performance relevant parameters like amount of I/O threads and queue blocking delay do not have the significant influence as expected, at least not for the overall throughput and response time. The reason is that their influence on e.g. the response time is not significant enough. Their influences are in the range of μs, whereas other components, e.g. the I/O interface are in the range of ms. Other factors identified during the calibration of the model have more significant influences on the system's performance (see figure 6.10), namely the CPU and DELAY resource demands of the I/O interface and the storage hardware.

- **DELAY resource demand.** The change of a component's demand on the DELAY resource (see figure 7.3) only influences the initial throughput of the system (which is the throughput under little load). It has no influence on the maximum throughput (achieved under high load). Hence, with less DELAY resource demand, the throughput-characteristics is moved to a higher initial throughput under little or medium load, implying that the saturation of the maximum throughput can be achieved earlier with less load. This observation can be explained by the fact that under little or medium load, the CPU is

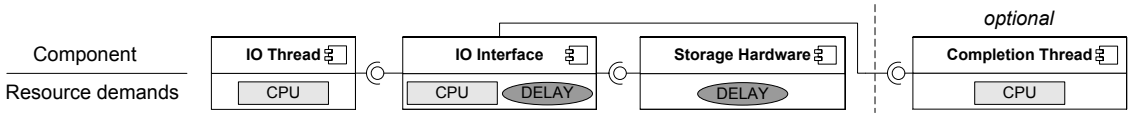


Figure 7.3: Component chain

not the bottleneck. Hence, shorting or prolonging the DELAY resource demand has a direct influence on the response time which in turn influences the throughput under little load.

- **CPU resource demand.** The CPU resource demand is influencing the maximum throughput of the system, only. Under little or medium load, the CPU resource demand has no influence on the initial throughput. In contrast, it massively influences the maximum throughput. This behavior is explainable as under full load the CPU is the system's bottleneck. The system's response times and hence the throughput are completely depending on the processing time. Thus, an additional CPU can have remarkable influences on the maximum throughput.

These influences observed during the model calibration lead to the following hypotheses about the influences of system components on the system's performance. They are particularly encouraged by the measurements with two CPUs. Further measurements with e.g. different storage hardware would be necessary to test these hypotheses.

1. The **I/O thread** is a CPU resource demanding component, only. Hence, its influence on the system's maximum throughput is observable, if its CPU resource demand is significantly higher compared to other resource demanding components. Then, an in/decrease of its resource demand can de/increase the system's maximum throughput. However, with the current configuration it has no significant influences.

2. The **I/O interface** is mainly consuming CPU resources. Its DELAY resource demands can be neglected for bigger requests, as the CPU resource demand dominates. But this CPU resource demand significantly influences the maximum throughput. Hence, optimizing this part of the system can provide a better system performance with respect to the maximum throughput.

3. The **storage hardware** is delaying the system's throughput. This has no influence on the maximum throughput under high load, whereas it significantly influences the throughput with few users. Hence, optimizing this system part can increase the system's performance for little or medium load, e.g. shorter response times for few users.

The CPUs assigned to the VL has significant influence on the throughput of the system. As the experiments with two CPUs show, the maximum throughput can be drastically increased. However, as the performance bottleneck CPU disappears, other limiting factors determine the performance, e.g. the bandwidth or throughput of the network and the response time of the storage hardware. Moreover, one should derive the maximum throughput from the kind of workloads issued to the system to determine how much CPU power is required.

8. Summary and Conclusions

The main objective of this work was to create a performance model of a potential storage virtualization for IBM systems. The model's purpose was to evaluate design alternatives of its architecture as a showcase on a System z. The implementation and configuration of the model was guided by the approach for an experiment-based derivation of software performance models by Happe. The target was to create and validate a performance model which reflects the behavior of a prototypical storage virtualization layer implementation, namely synchronous request handling. The synchronous model was validated by measurements of an existing system.

Furthermore, another objective was to model and predict the behavior of an asynchronous implementation. As no prototype existed for the asynchronous model, it was neither possible to configure nor to evaluate the asynchronous model by means of measurements of an implemented system. Hence, the asynchronous model was based on the expertise of synchronous model. Its evaluation is a discussion of the performance parameters' influences on the system's performance and a comparison with the synchronous model.

The selected modeling language was the Palladio Component Model (PCM), a domain-specific meta-model concentrated on component-based software architectures. Because it is not within PCM's domain, this work is also a study of PCM's applicability in other, PCM-foreign domains and whether it can meet IBM's requirements for a performance modeling tool.

During the modeling, several limitations of PCM appeared which required special solutions and additional modeling effort. For example, PCM does currently not support to parameterize the amount of a component's instance and abstracts from a component's state. This impeded the exact modeling of the request queues and I/O threads, and their queue access and locking behavior. Therefore, assumptions and abstractions had to be made to model all requirements.

Other difficulties arose because the simulation results contradicted the hardware measurements of the IBM system. Further analysis of the discrepancy of model and reality led to the revision of the measurement procedure and to new measurement results confirming the model's prediction. For example, the simulation of a

system with two CPUs predicted a higher throughput as measured on the reference system. The greatest problem was that the throughput measurements of the experiments contradicted the throughput measured while collecting the system component's response times. This resulted in a discrepancy of throughput measurements and simulation results. As a solution, the hardware measurements were only used to derive the influences of performance relevant factors, qualitatively.

The simulation results of the synchronous model show that the simulated behavior matches the experiment results with errors less than 10%. However, the validation of the synchronous model with a mixture of READ and WRITE requests did not yield the expected results of errors less than 10%. One explanation is that the measurements of the mixture might have been influenced, e.g. by cache hits, although the measurement tool was set to ignore caches. Another explanation is that the system might have READ/WRITE mix related properties not reflected by the model. However, the validation shows a good qualitative performance prediction of the synchronous model compared to the reference system.

Moreover, the evaluation of the model shows that the performance relevant factors like amount of I/O threads, amount of request queues and queue blocking do not have an impact on the overall throughput and only a slight impact on the response times. In fact, the main performance influence factors identified are the storage hardware and the I/O interface. The evaluation confirms the assumption of IBM that the performance of the asynchronous is not completely different from the synchronous implementation. Hence, the question whether synchronous is better than asynchronous I/O must be answered by trading off the general advantages and disadvantages of the two design alternatives.

Besides the performance modeling, this project was a study of the abilities and applicability of PCM. IBM has learned that it is possible to create a performance model with relatively low effort. Although IBM sees the supporting abilities of PCM in future development rather neutral, they confirmed the usefulness of PCM to improve the knowledge about a system. However, this work demonstrates that detailed knowledge is required to model a system with sufficient accuracy.

This modeling approach showed that PCM can be used for performance modeling in other domains besides component-based software architectures. However, then several limitations and restrictions apply. Hence, to be used in further projects, PCM must be extended by several features resolving PCM's current issues, some of them already familiar.

To extend the model of this work, further studies could conduct more detailed and complex measurements. Then, the results could be analyzed to better assess the validity of the model and to improve the quantitative reflection of the system behavior. Additionally, the full integration of the scheduler model by Happe could be used to analyze the scheduler's impact on the system's performance more detailed.

Furthermore, the identification of realistic workload profiles (e.g. a database workload) by IBM would be of interest. Then, the model could be used to determine which configuration suits best to which workload profile. Such a classification of workload types could help a system deployer to decide, how a system should be configured. Moreover, with typical workloads it might be possible to determine if one of the two design alternative is more suitable.

Glossary

Channel	A channel connects a system with the storage hardware. It has several properties like throughput and bandwidth constraints.
Client	A client resides a request-completion queue pair for each attached logical device. A client signals a new request for a device via the request queue. The actual request is put in the according request queue to be processed by an I/O thread.
Completion Queue	If an I/O operation of a request completes, the signal is put in the completion queue belonging to the request's origin.
Completion Thread	Processes the completion signal received from the OS and puts it in the corresponding completion queue. This thread not under control of the IOVF.
I/O Interface	The interface offered by the operating system to read/write data from/to logical devices. In this work, it is equivalent to the operating system the VL runs on.
I/O Thread	Processes the requests for a device signaled in a request queue by modifying and sending the request to the I/O Interface.
Logical Device	Logical representation of multiple physical devices or of a part of a physical device.
Partition (LPAR)	A logical partition (LPAR) is a virtual instance of the computer system identical to the hardware, but smaller. It can host several clients, e.g. guests running on a hypervisor like KVM.

Physical Device	The actual device.
Request	Specifies an I/O operation.
Request Queue	Stores the requests of the client for a device in the corresponding request queue.
Request-Completion Queue Pair	Each device assigned to a client has a Request-Completion queue pair. If a request out of the request queue completes, the completion is signaled in the corresponding completion queue.
Virtual Device	Virtual representation of a logical device.
VL	Virtualization Layer for I/O. Responsible for delivering requests from the clients' virtual devices to the physical devices and signaling the completion of these operations back to the client.

A. Measurements

The following contains detailed measurement and simulation results used for the experiments and the model calibration as well as simulation results of the evaluation chapter.

Request size/type	Request producers								
	1	2	4	8	16	32	64	128	256
4KB READ	6219	11766	23759	36815	39205	43297	46649	47539	47416
16KB READ	4707	8343	17091	28145	34584	36258	38493	39383	39518
64KB READ	2184	3424	6336	9605	11842	11622	11715	11780	11911
256KB READ	656	945	1886	1930	2824	2784	2929	3013	3053
1024KB READ	223	263	511	502	666	646	757	763	771
4KB READ, 2CPU	6327	11891	23905	42010	57052	68561	81891	87891	89230
16KB READ, 2CPU	4767	8899	17225	29958	49423	62205	62210	71338	69919
64KB READ, 2CPU	2324	3640	7079	11909	15709	18470	18774	18854	18789
256KB READ, 2CPU	706	912	2087	3244	3908	3984	3922	4002	4166
4KB WRITE	3276	6236	12736	23484	34262	35618	35254	35375	34487
16KB WRITE	2286	4483	8941	15717	24403	23965	24013	23450	23700
64KB WRITE	1130	2233	4496	7565	9847	9688	9574	9564	9553
256KB WRITE	498	914	1835	2613	2863	2874	2870	2902	2863
1024KB WRITE	175	233	480	723	714	721	710	712	723
4KB, 60/40% R/W Mix	3947	7725	13734	21983	33422	40927	45437	46857	46514
16KB, 60/40% R/W Mix	2887	5621	10009	18754	24277	34110	39618	40347	41150
64KB, 60/40% R/W Mix	1570	2462	4511	6497	9477	10635	12030	12465	13099
256KB, 60/40% R/W Mix	455	690	1235	1926	2208	3027	3146	3346	3332

Table A.1: Throughput measurements for different request sizes and types under varying load

Request Size/Type	1 User 1 CPU	1 User 2 CPUs	Relative Deviation	256 Users 1 CPU	256 Users 2 CPUs	Relative Deviation
4KB READ	6219	6327	1.02	47416	89230	1.88
16KB READ	4707	4767	1.01	39518	69919	1.77
64KB READ	2184	2324	1.06	11911	18789	1.58
256KB READ	656	706	1.08	3053	4166	1.36

Table A.2: Absolute throughput and relative deviation (2CPUs/1CPU) for little/high load (1/256 request producer(s))

Request		Initial Throughput				Maximum Throughput			
Type	Size	Measured	Simulated	Abs. Error	Rel. Error	Measured	Simulated	Abs. Error	Rel. Error
READ	4	6219	6049	170	2.73%	47416	48706	-1290	2.72%
	16	4707	4539	168	3.57%	39518	37879	1639	4.15%
	32	3580	3372	208	5.81%	22530	23196	-666	2.96%
	64	2184	2183	1	0.05%	11911	12889	-978	8.21%
	256	656	688	-32	4.88%	3053	3125	-72	2.36%
	1024	223	184	39	17.49%	771	820	-49	6.36%
WRITE	4	3276	3628	-352	10.74%	34487	36355	-1868	5.42%
	16	2286	2384	-98	4.29%	23700	22906	794	3.35%
	32	1690	1636	54	3.20%	15574	15338	236	1.52%
	64	1130	1221	-91	8.05%	9553	9144	409	4.28%
	256	498	543	-45	9.04%	2863	2593	270	9.43%
	1024	175	168	7	4.00%	723	717	6	0.83%

Table A.3: Absolute and relative errors for initial and maximum throughput of measurements and simulation for the final configured model

Resource demand setup	Request producers								
	1	2	4	8	16	32	64	128	256
as configured	688	1204	2090	2913	3192	3275	3262	3244	3227
30% less storage hardware delay	901	1490	2617	3151	3261	3281	3270	3232	3215
30% more storage hardware delay	556	980	1832	2885	3082	3254	3251	3260	3228
30% less I/O interface CPU demand	732	1344	2355	3930	4451	4553	4556	4577	4545
30% more I/O interface CPU demand	650	1066	1811	2286	2527	2556	2514	2484	2477

Table A.4: Simulation throughput results for different resource demand settings

R/W Mix	Initial Throughput				Maximum Throughput			
Req. Size	Measured	Simulated	Abs Error	Rel. Error	Measured	Simulated	Abs Error	Rel. Error
4	3947	4801	-854	21.64%	46.514	42.615	3899	8.38%
16	2887	3370	-483	16.73%	41.150	30.118	11032	26.81%
64	1570	1660	-90	5.73%	13.099	11.125	1974	15.07%
256	455	624	-169	37.14%	3.332	3.028	304	9.12%

Table A.5: Absolute and relative errors for initial and maximum throughput of measurements and simulation for READ/WRITE mix

READ request size	Initial TP					Maximum TP				
	Measured	2*Proc.Rate	Rel. Error	2 Cores	Rel. Error	Measured	2*Proc.Rate	Rel. Error	2 Cores	Rel. Error
4	6327	6454	2.01%	6219	1.71%	89230	96229	7.84%	90214	1.10%
16	4767	4890	2.58%	4691	1.59%	69919	75902	8.56%	71404	2.12%
64	2324	2383	2.54%	2204	5.16%	18789	25970	38.22%	24441	30.08%
256	706	768	8.78%	690	2.27%	4166	6485	55.66%	6218	49.26%

Table A.6: Absolute and relative errors for initial and maximum throughput of measurements with 2 CPUs and simulation with doubled CPU processing rate and exact scheduler with two cores.

			Throughput (requests/second) for x request producers								
ReqSize/Type	Queues	I/OThreads	1	2	4	8	16	32	64	128	256
256KB READ	1	1	3208	3208	3208	3208	3208	3208	3208	3208	3208
			3185	3185	3185	3185	3185	3185	3185	3185	3185
256KB READ	256	10	3208	3218	3236	3272	3290	3290	3290	3290	3290
			3185	3174	3152	3112	3090	3090	3090	3090	3090
256KB READ	= nr. of req.	10	3208	3218	3236	3272	3290	3290	3290	3290	3290
	producers		3185	3174	3152	3112	3090	3090	3090	3090	3090
256KB READ	256	100	3208	3218	3236	3272	3344	3488	3776	4100	4100
			3185	3174	3152	3112	3024	2848	2517	2200	2200
4KB WRITE	256	1	26757	26757	26757	26757	26757	26757	26757	26757	26757
			26749	26749	26749	26749	26749	26749	26749	26749	26749
4KB WRITE	256	10	26757	26757	26757	26760	26760	26760	26760	26760	26760
			26749	26748	26745	26736	26740	26740	26740	26740	26740

Table A.7: Amount of issued and completed requests of the asynchronous model for different parameter settings

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.

[And08] Roman Andrej. Evaluation des Vorhersageverfahrens "Palladio" im industriellen Kontext der CAS Software AG, 2008. Diploma thesis.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. PhD thesis, University of Oldenburg, 2008.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the Palladio Component Model. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 54–65, New York, NY, USA, 2007. ACM.

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

[CD00] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software (The Component Software Series)*. Addison- Wesley Professional, 2000.

[Dal08] Peter Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008. ISBN 978-0-387-79053-4.

[dev09] IBM developerWorks. Fibre Channel Protocol (FCP) statistics (SLES9 / SLES10). http://www.ibm.com/developerworks/linux/linux390/perf/tuning_how_dasd_scsiIO.html, 2009. Last changed: 2. April, 2009.

[ea07] Parziale et al. *Introduction to the new mainframe: z/VM Basics*. IBM redbooks, 2007.

[(EM09] Eclipse Modeling Framework Project (EMF). Homepage of the EMF Project. http://www.eclipse.org/modeling/emf/, 2009. Last changed: 2. April, 2009.

[Fri07] Holger Friedrich. Modellierung nebenlaeufiger, komponentenbasierter Software-Systeme mit Entwurfsmustern, 2007. Diploma thesis.

[Gre05] Wolfram Greis. *Die IBM-Mainframe-Architektur*. Open Source Press, 2005.

[Hap08] Jens Happe. *Predicting Software Performance in Symmetric Multicore and Multiprocessor Environments*. PhD thesis, University of Oldenburg, 2008. To be published.

[HGvdMH04] M. Harkema, B. M. M. Gijsen, R. D. van der Mei, and Y. Hoekstra. Middleware performance: A quantitative modeling approach. In *Symposium on Performance Evaluation of Computer Telecommunication Systems*, pages 733–742, 2004.

[HKW+07] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - the Common Component Modeling Example. In *CoCoME*, pages 16–53, 2007.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[KB07] Klaus Krogmann and Steffen Becker. A Case Study on Model-Driven and Conventional Software Development: The Palladio Editor. In *Software Engineering 2007 - Beiträge zu den Workshops*, volume 106 of *Lecture Notes in Informatics*, pages 169–176. Series of the Gesellschaft für Informatik (GI), 2007.

[Koz08] Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, 2008.

[KR08] Klaus Krogmann and Ralf H. Reussner. *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, chapter Palladio: Prediction of Performance Properties, pages 297–326. Springer-Verlag Berlin Heidelberg, 2008.

[Kro06] Klaus Krogmann. Entwicklung und Transformation eines EMF-Modells des Palladio Komponenten-Meta-Modells. Master's thesis, University of Oldenburg, Germany, 2006.

[KS08] Paul A. Karger and David R. Safford. I/O for virtual machine monitors: Security and performance issues. *IEEE Security and Privacy*, 6(5):16–23, 2008.

[LHAP06] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, page 3, Berkeley, CA, USA, 2006. USENIX Association.

[Lit61] John D. C. Little. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[Men05] Daniel A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conference*, pages 407–414, 2005.

[PGGG06] U. Praphamontripong, S. Gokhale, Aniruddha Gokhale, and Jeff Gray. Performance analysis of an asynchronous web server. *Computer Software and Applications Conference, Annual International*, 2:22–28, 2006.

[PGGG07] U. Praphamontripong, S. Gokhale, Aniruddha Gokhale, and Jeff Gray. Performance analysis of a middleware demultiplexing pattern. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 287a, Washington, DC, USA, 2007. IEEE Computer Society.

[RBK+07] Ralf H. Reussner, Steffen Becker, Heiko Koziolek, Jens Happe, Michael Kuperberg, and Klaus Krogmann. The Palladio Component Model. Interner Bericht 2007-21, Universität Karlsruhe (TH), 2007.

[Reu01] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[RG05] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, 2005.

[RP00] Sridhar Ramesh and Harry G. Perros. A multilayer client-server queueing network model with synchronous and asynchronous messages. *IEEE Trans. Softw. Eng.*, 26(11):1086–1100, 2000.

[RS07] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188, New York, NY, USA, 2007. ACM.

[SHB07] Douglas Schmidt, Kevlin Henney, and Frank Buschmann. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley, 2007.

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[Ufl05] Matthias Uflacker. Design of an editor for the model-driven construction of component based software architectures. Master's thesis, University of Oldenburg, 2005.

[VNOS08] Geoffroy Vallee, Thomas Naughton, Christian Engelmannand Hong Ong, and Stephen L. Scott. System-level virtualization for high performance computing. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 636–643, Washington, DC, USA, 2008. IEEE Computer Society.

[WJW07] Jinpeng Wei, Jeffrey R. Jackson, and John A. Wiegert. Towards Scalable and High Performance I/O virtualization - A Case Study. In *HPCC*, pages 586–598, 2007.

[WRJ07] J. Wiegert, G. Regnier, and J. Jackson. Challenges for scalable networking in a virtualized server. *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 179–184, Aug. 2007.