Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Extending the Palladio Meta-Model to Support Memory Hierarchy

Kim Truong

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Markus Frank, M.Sc.
Commenced:	March 2, 2020
Completed:	October 28, 2020

Abstract

Poor software architecture design decisions can lead to software performance issues. Often these performance issues are detected when code is already written. Therefore, the architecture must be redesigned and code must be reimplemented, which is costly. With a model-based performance prediction approach, software architects can detect potential performance bottlenecks at an early stage of the software development lifecycle. One such approach is the Palladio approach. The Palladio approach offers tools that use model-based simulations to predict performance metrics such as response time. However, Palladio can not give accurate predictions for multicore systems because it currently neglects the scaling limitations of multicore systems. One such limitation is the memory hierarchy and its bandwidths, which is a performance bottleneck for parallelized applications. In this thesis, we extended Palladio to support the modeling and simulation of the memory hierarchy (e.g., the influence of L1-, L2-, L3-cache, DRAM, and their bandwidths). To identify all necessary elements that affect the prediction accuracy, we used an experiment-based approach. We stepwise integrated and evaluated influencing factors related to memory hierarchy and its bandwidth until we achieve a good response time accuracy. To evaluate the prediction accuracy, we use the proposed extension to model and simulate a parallelized matrix multiplication implementation in Palladio. The simulated response time showed a mean prediction error in a range of 6.5% to 50.3% for our final data transfer model. The results indicate that by including the memory hierarchy and its bandwidth, the response time prediction is more accurate for cases in which the connection to the main memory is saturated.

Kurzfassung

Falsche Architekturentscheidungen können zu Software-Performance-Probleme führen. Oft werden diese Performance-Probleme erst entdeckt, wenn bereits Code geschrieben wurde. Es entstehen dadurch neue Kosten, weil die Architektur und der Code neu angepasst werden müssen. Mit einem modellbasierten Ansatz können Softwarearchitekten Leistungsengpässe bereits in einer frühen Phase des Software-Lebenszyklus entdecken. Ein solcher Ansatz ist Palladio. Der Palladio-Ansatz bietet Werkzeuge an, die modellbasierte Simulationen nutzen, um Vorhersagen über Performance-Metriken wie die Antwortzeit zu treffen. Allerdings kann Palladio keine genauen Vorhersagen für Multiprozessorsysteme treffen, da Palladio derzeit die Limitationen der Skalierbarkeit von Multiprozessorsystemen nicht berücksichtigt. Eine der Limitationen ist die Speicherhierarchie mit ihren Bandbreiten, die ein Leistungsengpass für parallelisierte Anwendungen ist. In dieser Arbeit wird Palladio erweitert, um die Modellierung und Simulierung der Speicherhierarchie (z.b. der Einfluss von L1-, L2-, L3-Cache, DRAM und ihren Bandbreiten) zu ermöglichen. Um alle nötigen Elemente, die einen Einfluss auf die Genauigkeit der Performance-Vorhersage haben, zu identifizieren, wird ein experimentbasierter Ansatz verwendet. Dabei werden schrittweise neue Einflussfaktoren, die der Speicherhierarchie zugehörig sind, integriert und ausgewertet. Um die Genauigkeit der Performance-Vorhersage auszuwerten, wird in Palladio, mithilfe der entwickelten Erweiterungen, eine parallelisierte Matrixmultiplikation modelliert und simuliert. Das finale Datentransfermodel hat für die vorhergesagte Antwortzeit einen durchschnittlichen Vorhersagefehler, der zwischen 6.5% und 50.3% liegt. Die Ergebnisse weisen darauf hin, dass durch das Hinzufügen der Speicherhierarchie mit ihren Bandbreiten die Vorhersage für die Antwortzeit bei fällen, in denen die Verbindung zum Arbeitsspeicher ausgelastet ist, genauer geworden ist.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

DRAM Dynamic random-access memory. 13

EMF Eclipse Modeling Framework. 7

GMF Graphical Modeling Framework. 11

HDD Hard disk drive. 13

LLC Last Level Cache. 17

PCM Palladio Component Model. 1

QoS Quality of Service. 9

QPI QuickPath Interconnect. 12

SEFF Service Effect Specification. 20

SPE Software Performance Engineering. 8

SRAM Static random-access memory. 13

1 Introduction

Late detected software performance issues caused by wrong architecture decisions result in costly redesigns and reimplementations. The software architecture is designed at an early stage and sets the foundation for the structure and behavior of the software system. A wrong design may result in performance problems that remain undetected until the test phase or even after launching the system. The consequence of late detected performance problems is costly redesigns and reimplementations. This additional work increases the time and the cost budget. Furthermore, a delayed software release or performance problems after release might result in reputation loss. A possible countermeasure would be to use a model-based performance prediction, which can detect potential performance bottlenecks at an early stage of the software development life-cycle [WFP07]. One model-based approach is the Palladio approach [RBH+16], which has been researched for over 10 years and it is still actively researched until now. Furthermore, Palladio has been used in various industrial projects [1]. With the Palladio approach, architecture can be simulated to predict quality attributes such as performance or reliability. One core part of the Palladio approach is the Palladio Component Model (PCM)[BKR09; RBB+11], which is a meta-model that was developed to specify component-based software architectures. PCM specifies different submodels, which are required to run a simulation. For example, PCM offers models to describe software, hardware, and usage profile. A model specifying software can specify, which functions can be called and how many resources (e.g., CPU) each call requires when used. A model specifying hardware describes available computer nodes with their provided CPU speed. Furthermore, computer node connections can be modeled with network links, which can have throughput and latency specifications. PCM also specifies a model that can specify the usage profile, e.g., how simulated users would interact with the system. Based on the above-mentioned models, model-based simulations can be used to predict concrete values such as response time.

1.1 Problem Statement

PCM was designed over 10 years ago with limited support for concurrency predictions [BKR09]. Frank et al. [FH16] showed that the performance predictions are not accurate when multicore systems are modeled with Palladio, because Palladio's prediction model assumes a nearly perfect linear speed-up (e.g., 100 cores means 100 times faster). However, Frank et al. demonstrated with an experiment that the real measured execution does not have a nearly perfect linear speed-up and that the accuracy of the prediction is only at 79% for a 16 core usage. In [FBKK19; FKB18] Frank et al. assume that the memory hierarchy and memory bandwidth might have an influence on the prediction accuracy and that it is necessary to model them.

[1] https://www.palladio-simulator.com/consulting/references/

Currently, the PCM meta-model, which specifies what elements can be modeled in Palladio, misses elements to model memory hierarchies, e.g., main memory, L1-, L2- and L3-cache. Therefore, memory hierarchy effects such as the costs of cache misses are neglected. Also, the concept of memory bandwidth, which is the rate of data that can be transferred through the memory hierarchy, is missing. In the case of multicore CPUs, memory bandwidth effects play an important role. Because the memory bus has a fixed and limited bandwidth, which is shared by multiple cores. Furthermore, in an experiment by Gruber [Gru19] it was shown that the data transfer through the memory hierarchy can be imitated with Palladio network link elements. His results indicate that the memory hierarchy has an impact on the accuracy of the response time prediction. For this reason, it is necessary to include the influence of memory hierarchy and memory bandwidth to the PCM meta-model.

Goals and Research Questions

The goal of this thesis is to achieve more accurate performance prediction with Palladio for multicore systems. Thereby, we focus mainly on the quality attribute response time. We assume that memory hierarchy and memory bandwidth affect prediction accuracy. To reach the above-stated goal, we posed the following subgoals:

G1 More accurate performance prediction with Palladio for multicore systems with a focus on response time.

- **G1.1** The PCM meta-model must be extended so that the influence of the memory hierarchy and memory bandwidth on the performance can be modeled.
- **G1.2** The simulation tool (SimuLizar) must be adapted to simulate both influences.
- **G1.3** The Sirius-based Palladio editors must be adapted so that the newly introduced modeling elements can be graphically modeled.

To achieve the above-stated goal and its subgoals, the following research questions are investigated:

RQ1 What elements are missing to model memory hierarchy and memory bandwidth?

RQ2 How can the PCM meta-model be extended with the identified elements?

- RQ2.1 Can existing PCM elements be reused?
- RQ2.2 What are the trade-offs between the different ways to extend a meta-model?

RQ3 How can the Palladio tools be adapted to the extended PCM meta-model?

- RQ3.1 How to adapt the simulation tool SimuLizar?
- RQ3.2 How to adapt the Sirius-based Palladio editors?

RQ4 Does this extension improve the accuracy of multicore performance prediction compared to previous works?

1.2 Solution Approach

To tackle the problems addressed in the previous section, we divided the solution process into three phases. Figure 1.1 shows the general approach with the three phases: overview, meta-model extension, and implementation. In the overview phase, we gather information about the necessary elements to model the memory hierarchy and memory bandwidth. The first entry point is Gruber's bachelor thesis [Gru19] in which we identify, characterize, and extract necessary elements. Additionally, we conduct a literature review to gain an overview of the memory hierarchy and possible elements that should be integrated. In the second phase, we propose and evaluate meta-model extensions alternatives and the most promising alternative is used. In the last phase, we adapt Palladio tools to the extended PCM meta-model. Thereby, we adapt Palladio editors and the simulation tool SimuLizar to the new introduced model elements. Furthermore, we validate the predicted results with the use case from [FH16] and [Gru19]. In addition, we test and integrate additional performance influencing factors of the memory hierarchy into Palladio in several iterations to increase the accuracy. For this, we use the experiment-based performance model derivation method by Happe [pp. 44-49][Hap09], which is described next.

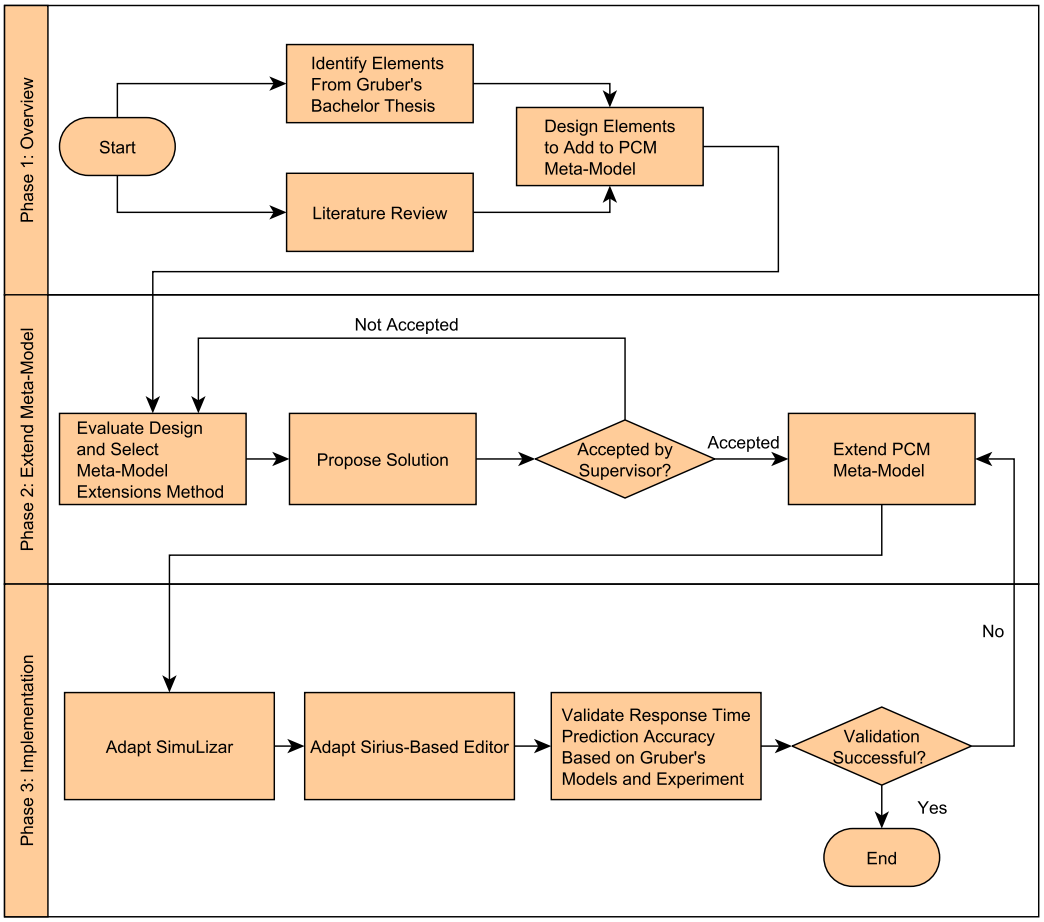


Figure 1.1: Flowchart of solution approach

Having described the general solution approach, we will now describe how we plan to design a meta-model that is suited to achieve a better response time prediction. For creating an initial meta-model of the memory hierarchy, we start with the goal-driven approach to model quality attributes described by Koziolek in [RBH+16, Chapter. 5], which helps to decide what elements should be modeled and what to omit. More details of this approach are presented in Section 4.2.1. For creating a performance model, we use the experiment-based performance model derivation method by Happe[pp. 44-49][Hap09], which is more specific than the approach described by Koziolek. This performance model derivation process describes how to test new influencing factors and how to integrate them into a performance model. Furthermore, this process is also driven by specific goals, for example, prediction accuracy or generalizability (e.g., the model should be flexible for multiple use cases). Happe's process consists of 5 steps, which are executed iteratively. Each step can be repeated with new insights to refine the performance model. The 5 steps from Happe's process (see Figure 1.2) are described as follows:

Identification: The first step tries to identify performance-relevant factors with respect to the initial goal of the model. This step is not trivial for cases in which multiple factors influencing each other, e.g., some factors only influence the performance if they occur at the same time. These initial influencing factors could either be retrieved from documentation or functional(specification) of the system. However, other methods such as a systematic literature review or expert interviews as described in [FBKK19] are also possible.

Experiment Design: In this step, an experiment is designed to systematically evaluate, the previously listed assumptions and influences. In this step, a concrete use case should be described in which some influencing factors are evaluated. Furthermore, necessary metrics that contribute to the previously defined goal are defined.

Experiment: Based on the experiment design, an experiment is conducted to validate or invalidate previous assumptions. If the measurements of the previously defined metrics support the assumptions, then the factors can be integrated into the performance model. However, the integrated factors can be still removed afterward, if the assumptions proved to be wrong afterward.

Performance Model Design: In this step, factors, which are based on valid assumptions from the experiment, can be used to design a prediction model. Happe also mentioned that the designed model in this stage are abstractions of the specific use case, and might only be correct for that specific use case.

Model Validation: In the last step, the designed prediction model must be validated, e.g., is the prediction accuracy is good enough or not? Furthermore, the limitations and generalizability of the performance model should be checked. Happe also suggests that there should be a balance between accuracy and flexibility for other use cases, e.g., that a lower accuracy is somehow acceptable if the performance model can also be used for other use cases.

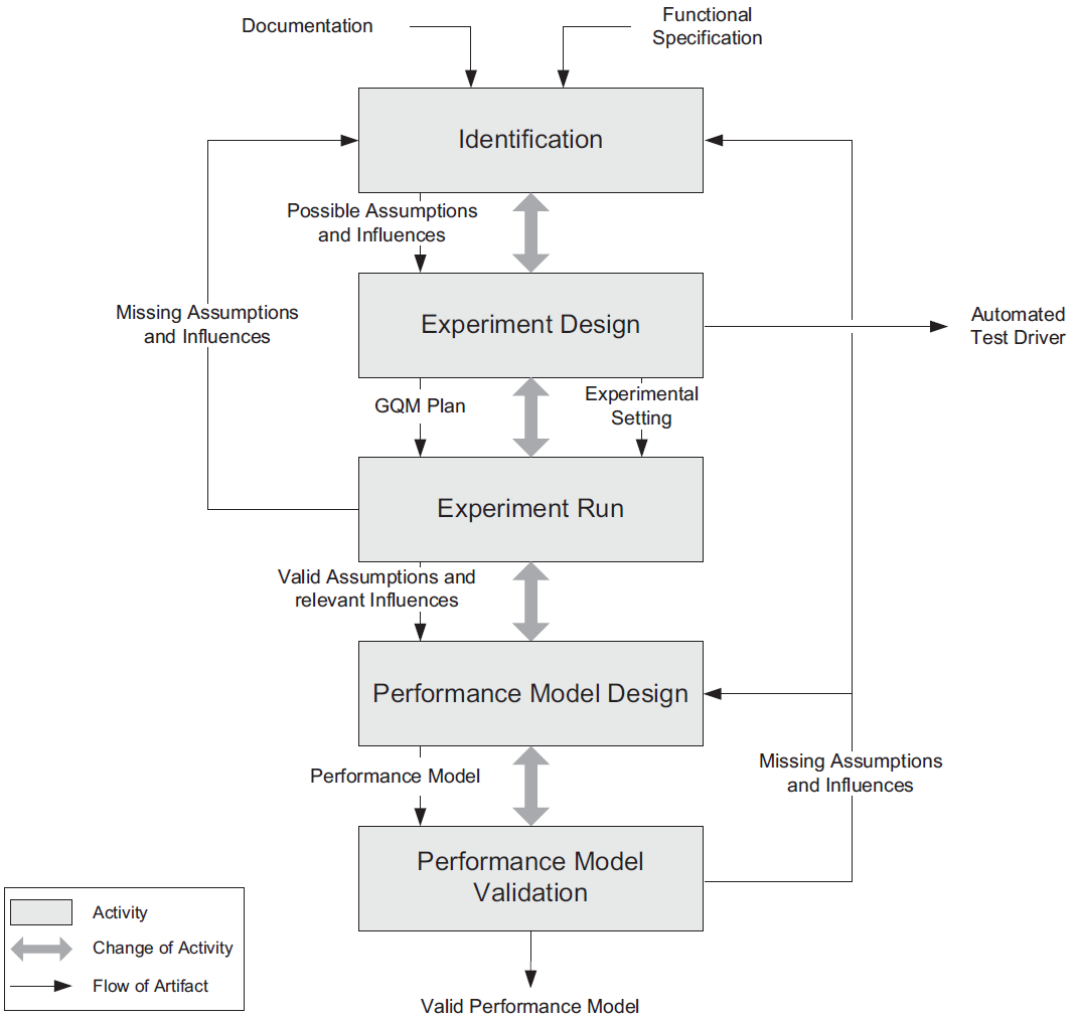


Figure 1.2: Experiment-based performance model derivation approach by Happe [Hap09, p. 45]

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 2 – Foundations describes the foundations this thesis is based on.

Chapter 3 – Related Work: In this chapter, the related work of this thesis are listed and discussed.

Chapter 4 – Meta-Model for Memory Hierarchy: In this chapter, the modeling process and the initial meta-model of the memory hierarchy are described. Furthermore, existing concepts and elements of PCM are evaluated for possible reuse.

Chapter 5 – Integrating PCM Extension in Palladio Toolchain describes the adaption of Palladio tools such as SimuLizar or Sirius-based editors.

Chapter 6 – Experimental Evaluation: In this chapter, the evaluation process, experiment setup, and Palladio model calibration are described. Furthermore, Palladio model concepts with different influencing effects of the memory hierarchy are evaluated and interpreted.

Chapter 7 – Conclusion: In the last chapter, the limitations, lessons learned, and future work are discussed. Furthermore, the research questions of this thesis are answered.

2 Foundations

This chapter introduced the foundations and technical background that are necessary to understand this thesis. A definition of the terminology of model and meta-model is given in Section 2.1. Additionally, a technical framework for modeling is briefly introduced. In Section 2.2 the term software performance engineering is defined, and some basic concepts of it are explained. Then, a broad overview of Palladio in Section 2.3 is given. Finally, in Section 2.4 the foundations of computer architecture, the memory hierarchy, and performance measurement tools are introduced.

2.1 Model and Meta-Model

In this section, the terminology of a model and a meta-model is defined. Furthermore, a technical framework that supports model creation for applications is introduced.

Based on Stachowiak [Sta73, pp. 131-133], a model must have at least the following three main features. As Stachowiak's original work is in German, we follow the translation by naming and meaning from the Palladio book [RBH+16, p. 39]:

Representation: A model represents an original. An original could be an entity in the real world or even a model itself, e.g., a model of a model.

Reduction: A model commonly does not have all properties of the original. Only the relevant properties of the original are represented.

Pragmatic: A model should serve a specific purpose, e.g., a model can be used for documentation or communication purposes, or a model can be used to model software architecture with the purpose of performance evaluations.

From the Palladio book [RBH+16, p. 40] a meta-model is defined as:

Definition 2.1.1
A meta-model is a model that represents models. The models that are represented by a certain meta-model conform to this meta-model. A meta-model serves the purpose of defining properties that conforming models can possess, and the rules that conforming models have to obey.

We further introduce the Eclipse Modeling Framework (EMF) [SBMP08], which provides a framework for modeling. EMF further provides code generation, which can generate concrete Java classes from a previously specified data model. The data model, for example, can be a domain model and describe all necessary Java classes, which are later used to develop an application. Furthermore, EMF can also generate editors in which an instance of the defined data model can be edited, e.g., it is possible to create elements of the data model in a tree view and edit properties of them.

2.2 Software Performance Engineering

In the definition of Woodside et al.[WFP07] Software Performance Engineering (SPE) is described as "*activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements*". Woodside et al. listed these activities, some basic activities are for example:

- the identification of performance requirements;
- defining scenarios, which describes a step by step behavior of software in a specific use case;
- definition of an operational profile, which describes how often operations are called and used:
- performance modeling, which maps performance affecting factors to a model;
- performance model analyzing, which analyzes the performance model to get performance prediction, e.g., throughput or computing time;
- run time monitoring of live software performance; and
- take performance measurements.

The authors distinguished between two approaches in SPE. The measurement-based approach, which happens in the later development cycle in which software can be run and measured, and the model-based approach, which happens in early development phases. A benefit of the model-based approach is that it can be used to predict performance before code is written. Furthermore, it is possible to predict the impact of design changes with models without having to implement them. Additionally, models can be refined in the later phases to provide a more accurate prediction. However, detailed models that cover every performance factor can be costly to model and it is costly to analyze these detailed models to get performance predictions. However, the performance predictions are just approximations of the real performance measure. On the other hand, the measurement-based approach gives accurate measurements in real environments and this approach might also detect additional performance problems, which were not detected in the model-based approach because the model did not cover every performance influencing factor. However, in the model-based approach, performance problems are detected at a later development phase, and therefore a redesign is costly. Both approaches are not mutually exclusive and can be combined to benefit from their strengths.

2.3 Palladio

Palladio provides tools to model and analyze software architectures. In the following sections, the modeling abilities and tools are introduced.

2.3.1 Palladio Modeling

A core element of Palladio is the Palladio Component Model (PCM)[BKR09], which provides the meta-model to specify software architectures. Furthermore, PCM focuses on Quality of Service (QoS) attributes, such as performance and reliability. Initially, Palladio was designed to model and analyze business information systems, however, it can be used in other domains as well [RBH+16, p. 9]. PCM modeling supports the component-based software engineering development process in which 4 roles are distinguished: component developer, system architect, system deployer, and business domain expert [BKR09]. In [RBH+16, pp. 23-24] it is described that the PCM consists of the following five main submodels:

Component Repository Model: The Component repository is normally modeled by a component developer. The component modeler models the functionality of a component and its resource consumption. Furthermore, interfaces of a component can be modeled, which defines which functionality is provided to other components.

System Model: The System Model is normally modeled by a system architect. The system architect models how different components are connected. Therefore, he is also able to swap out different components so that alternative architectures can be analyzed and compared.

Execution Environment Model: The Execution Environment Model is normally specified by a system deployer. In this model, the available server nodes are modeled. For example, it is possible to specify the resource processing abilities of a node. Furthermore, network links with their throughput and latency can be specified to model connections between server nodes. In the rest of this thesis, this model is also referred to as resource environment model.

Component Allocation Model: The Component Allocation Model is normally specified by a system deployer. The system deployer has the most knowledge about the available hardware, therefore, he can decide how to assign components to a suitable server node.

Usage Model: The Usage Model is normally specified by a business domain expert. The business domain expert specifies how users would interact inside the modeled system. For example, the business domain expert can specify how many or how often users want to access a specific functionality of the system.

A general overview of the different models is shown in Figure 2.1. Over time PCM was extended to support additional models, for example, QoS monitors, which specifies which QoS should be measured during the analysis.

2.3.2 Palladio Tools

This section gives an overview of some provided simulation tools of Palladio.

Palladio Performance Analysis Tools

Palladio offers different tools to analyze a system that is described by the models from the previous section. Based on the requirements of the performance analysis, different analysis tools can be selected. Factors like total analysis run time or the complexity of the models play a role in the

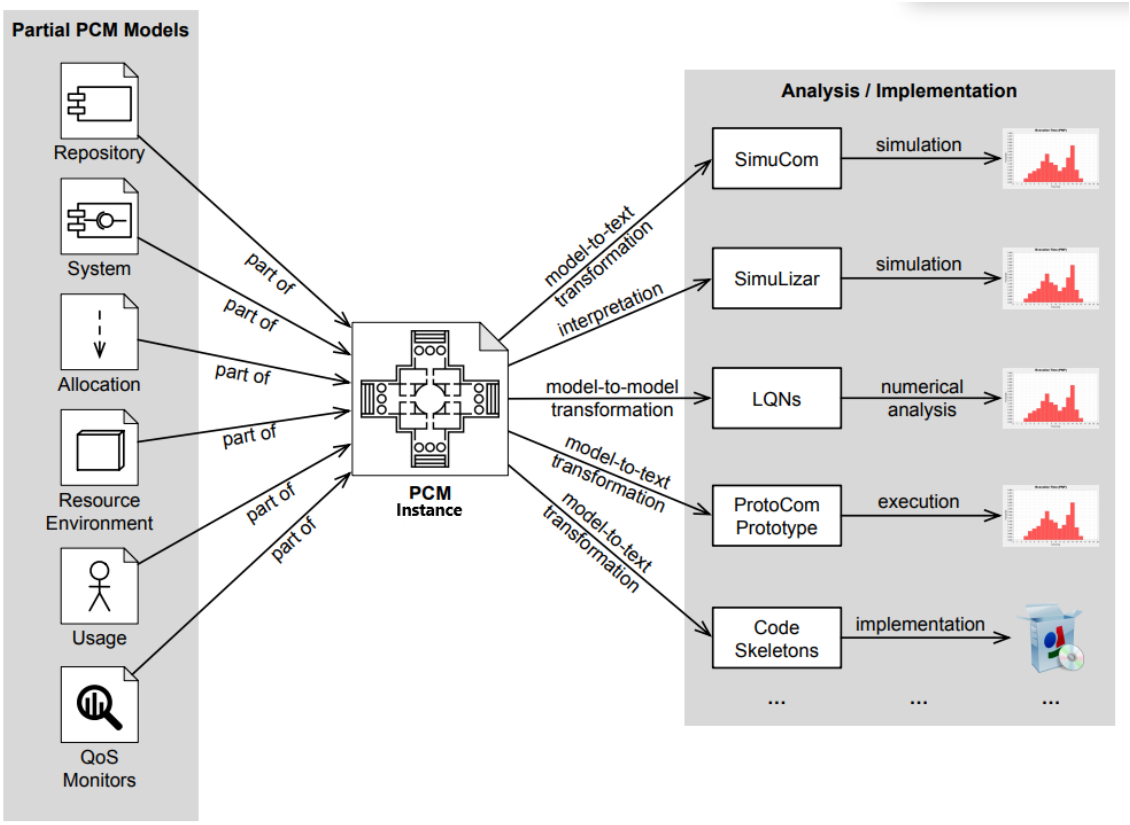


Figure 2.1: Overview of a PCM instance. This figure is modified from [Leh18, p. 73]

decision. Brosig et al. [BMB+14] described the trade-off between the various tools. Furthermore, Brosig et al. provide a decision tree to help on the selection of the most appropriate analysis tool. In the Palladio book [RBH+16, pp. 183-185], the decision tree is updated with the two analysis tools SimuLizar and EventSim. Additionally, the Palladio book recommends to use model solvers first and then validate the results with ProtoCom. ProtoCom [Bec08, pp. 196-200] transforms Palladio models into a performance prototype that mimics the demand on real hardware resources. This performance prototype in form of Java code then can be executed in a real environment and measured to gain more confidence about the predicted results. However, the setup and execution of the prototype take much more time than the model, for example, the Palladio book mentioned that the time required to set up and run ProtoCom can take up to several hours, whereas model solvers only require a few minutes.

Before introducing some performance prediction tools, a few terms are described to understand this section better. The terms are:

PCM instance: In this thesis, we will describe a PCM instance as a set of models, e.g., the five models(usage, system, repository, allocation, and resource environment) for a specific system that should be analyzed for performance.

Discrete-Event Simulation: A discrete-event simulator consists of at least two components a simulation timeline and events. Events can programmatically be scheduled and also progressed further on the timeline. This is just a broad description of discrete-event simulation for in deep knowledge we refer further to [PKG05].

Simulation-based solvers: We will refer to them as solvers, which analyzes a PCM instance by simulation, e.g., by using a discrete-event simulator. However, Palladio also offers analytical-based solvers.

In the following two performance prediction tools, which are required to understand this thesis, are described:

SimuCom: SimuCom [Bec08, pp. 127-128] is a simulation-based solver and consists of two parts. Part one can generate Java code from a PCM instance with model-to-text transformation. This generated code can then be executed to start the simulation. Part two is the SimuCom platform, which contains the simulation logic that is used by the generated Java simulation code. This SimuCom platform is not purely used by the generated simulation code, but also by other simulators, such as SimuLizar because it contains reusable simulation code. Furthermore, the process of Java simulation code generation and its execution is an automated process, e.g., the performance analyst just has to launch a SimuCom analysis run to gather the simulated QoS metrics and the simulated code is generally not visible to him. At the time of writing this thesis, SimuCom is deprecated and therefore the structure and naming of SimuCom might change in the future.

SimuLizar: SimuLizar [BBM13; Bec17; Mey11] is also a simulation-based solver. Additionally, it supports the analysis of self-adaptive systems, e.g., systems that should scale dynamically when the workload changes or service level objectives are violated. To analyze self-adaptive systems, PCM is extended by a few further models. These models, for example, can specify monitoring and self-adaption rules. In contrast to SimuCom, the SimuLizar simulator does not generate simulation code. SimuLizar follows an interpreter-based approach instead. Meyer [Mey11, pp. 38-40] argues that a generator-based approach is faster for non-adaptive systems, however, for adaptive systems the generative approach is unsuited because the generated code must be modified each time an adaption occurs. In the interpreter-based approach, the simulator traverses through a PCM instance and interprets the encountered model elements. SimuLizar still reuses the SimuCom platform with its simulation logic to simulate a PCM instance.

Graphical and Tree Editors

The Palladio tools offer graphical editors based on the Graphical Modeling Framework (GMF)[1] or the newer graphical editors based on Sirius[2]. Furthermore, Palladio provides tree editors, which are generated by EMF.

[1] https://www.eclipse.org/gmf-tooling/
[2] https://www.eclipse.org/sirius/

2.4 Computer Architecture

The following sections describe multicore systems. First, the architecture of multicore systems is introduced. Then, the memory hierarchy is introduced, and finally, some performance measurement tools are introduced.

2.4.1 Multicore Architecture

There are various multicore architecture design approaches that try to increase the number of usable cores. In this section, the focus lies on multicore systems with multiple processors. Figure 2.2 describes an exemplary server. The server consists of 2 processors each with 6 physical cores. Most servers also allow Hyper-Threading, which enables that 2 threads can run in parallel on a single physical core. However, these two threads on the same core have to share some processing resources such as access to caches [Dre07]. These additional threads are also referred to as logical or virtual cores. For example, the server described in Figure 2.2 has 12 physical cores and with Hyper-Threading activated 24 virtual cores. Furthermore, processors are connected with processor specific interconnects (e.g., QuickPath Interconnect (QPI) for Intel processors or HyperTransport for AMD processors). In Figure 2.2 the red dashed lines represent possible performance bottlenecks. These bottlenecks are caused by shared resources.

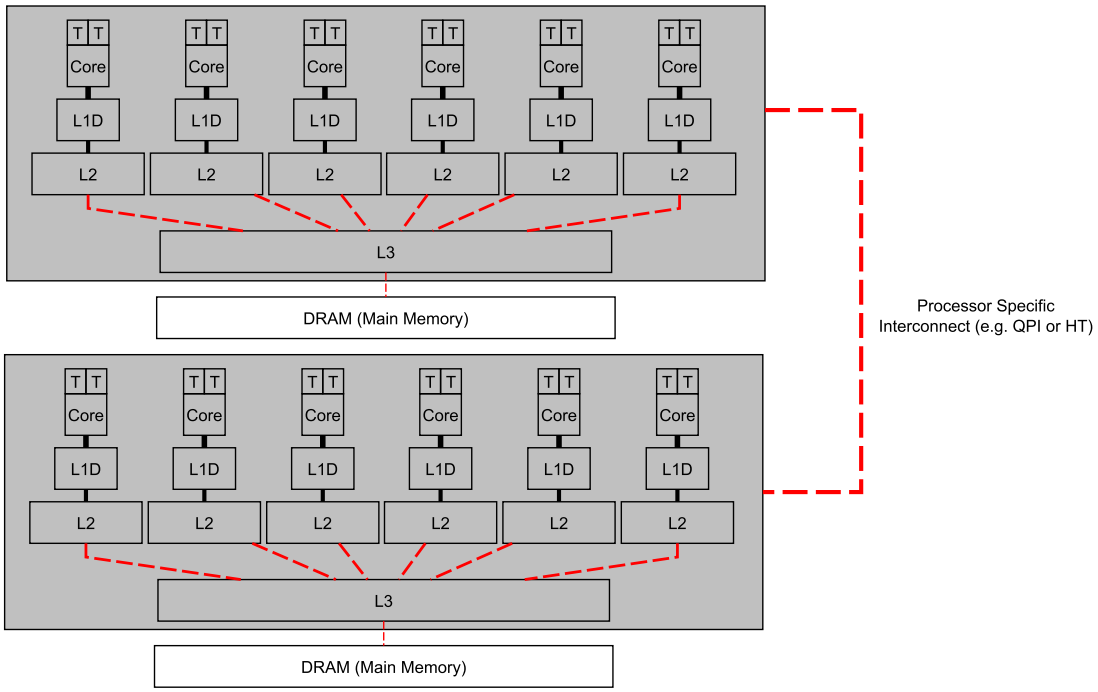


Figure 2.2: An example of a machine with two CPUs with 6 Cores and Hyper-Threading. The smaller blocks on top of the cores represent the virtual cores, e.g., each physical core is able to handle two threads simultaneously. The HT is an abbreviation of AMD's HyperTransport. This figure is based on [Hag, p 22]

2.4.2 Memory Hierarchy

The following section about the memory hierarchy is based on the book from Patterson et al. [PH16, Chapter 5]. It is optimal for performance when a processor can access the required data from memory as fast as possible. Currently, different memory types exist, e.g., Dynamic random-access memory (DRAM), Static random-access memory (SRAM), or Hard disk drive (HDD). Each memory type has a different data access time, for example, SRAM has a shorter access time than DRAM. However, the price of the SRAM is much more expensive than DRAM. Therefore, using a lot of SRAM is economically not feasible. Memory hierarchies are introduced to provide fast data access while still being economically efficient. A memory hierarchy uses multiple levels of memories with different memory technologies, e.g., caches use SRAM and the main memory uses DRAM. A hierarchical cache design is shown in Figure 2.2, which shows the L1D and L2 as private caches, and L3 as a shared cache. Normally a memory hierarchy also has the L1I instruction cache, however, this was not included in Figure 2.2. The memory hierarchy exploits the locality of reference (also known as the principle of locality), which describes how computer programs typically access data. There are two types of locality:

- **Temporal locality (locality of time)**: If a memory location is referenced, then there is a high possibility that the memory location is referenced again in a short time. For example, many programs have a loop in which a variable is modified over and over again.
- **Spatial locality (locality in space)**: If a memory location is referenced, then there is a high possibility that a close memory location will be referenced next. For example, programs have lists or arrays and often items from these lists or arrays are accessed sequentially.

These types of locality can be used to predict frequently used data, and store them into the upper levels of the memory hierarchy.

Here, some further important aspects of caches are introduced such as:

Cache associativity: This describes how a specific cache is structured, e.g., how and where data can be stored inside a cache.

Cache miss rate: The hit or miss rate is dependent on various other factors such as cache associativity, cache replacement strategy (e.g., which cache block should be removed when the cache is full).

Cache line size: This describes that caches have a specific size of a unit in which data is transferred through the cache levels. The values are usually between 16 and 128.

Cache inclusion: Lower cache levels can be inclusive or exclusive. For example, exclusive L3 cache means that data that is already stored in higher cache levels such as L1 or L2 is not also in L3. An inclusive cache allows the storage of the same data in multiple cache levels.

2.4.3 Performance Tools

In this section, different monitoring tools are presented.

Hardware Performance Counter

Modern processors have inbuilt hardware counters that can be used to measure or debug their behavior. Most of the measurable hardware events may differ on different processor architectures, e.g., it must be checked whether the event is available on the processor or if they really measure the desired information. Therefore, it is recommended to look up available hardware events from the specification, e.g., the available performance counters for Intel processors are listed in volume 3 and chapter 19 of the Intel developer manual [Int20, Volume 3, Chapter 19]. Perf is a tool that can measure hardware counters via the command line. However, perf is restricted to Linux based operations systems. Perf provides the *perf list* command to list all available named hardware events on the machine. These named hardware events can then be used with the *perf stat -e* command to get measurements. For example, the *perf stat -e L1-dcache-load java -jar Application.jar* command would execute the *Application.jar* file and measure all occurrences of the *L1-dcache-load* event, which represents L1D cache accesses. Most machines provide all the necessary hardware counters to measure the memory hierarchy. A further advantage of perf is that it is preinstalled on newer Linux versions.

STREAM

The STREAM benchmark [McC+95] can measure the memory bandwidth for the four simple vector kernels COPY, SCALE, SUM, and TRIAD (see table 2.1).

Name	Kernel
COPY	a(i) = b(i)
SCALE	a(i) = q*b(i)
SUM	a(i) = b(i) + c(i)
TRIAD	a(i) = b(i) + q*c(i)

Table 2.1: Vector kernels used in STREAM

By executing the STREAM Benchmark, the bandwidth to the DRAM is returned for the four vector kernels. Furthermore, STREAM is also able to measure the adapting memory bandwidth for multicore machines. The STREAM implementation is provided in the two languages C and Fortran. The STREAM implementation relies on OpenMP to parallelize the code for multicore measurement (further details can be found on the STREAM website [3]). In Listing 2.1 an exemplary excerpt from the result from a STREAM measurement is shown.

[3] https://www.cs.virginia.edu/stream/ref.html

Listing 2.1 Excerpt of results from STREAM benchmark with 4 threads

```
Number of Threads requested = 4
Number of Threads counted = 4
-------------------------------------------------------------
...
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           21075.9     0.124504     0.075916     0.314559
Scale:          21463.8     0.103438     0.074544     0.209747
Add:            24286.1     0.135530     0.098822     0.237762
Triad:          24373.2     0.125379     0.098469     0.213929
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
```

MemTest86

MemTest86[4] is an application to detect faults in the memory of a system. It is available for common operating systems such as Windows, Linux, and macOS. Memtest86 can also be used to measure the bandwidths between the various caches, e.g., the bandwidths for core-L1, L1-L2, L2-L3, and L3-DRAM.

[4] https://www.memtest86.com/

3 Related Work

The starting point to find related work is the systematic literature review by Frank et al. [FHLB17], which was conducted in the year 2017. We used the snowballing approach and focused on studies that are published after 2017. Furthermore, the focus is mainly on approaches that evaluated machines with more than 8 cores. We also used Google Scholar to get a deeper understanding of the scaling problems of the matrix multiplication on multicore systems. The keyword used for this is (“matrix multiplication”) AND (“amdahls law” OR “multicore scaling” OR “scaling”).

3.1 Performance Prediction for Multicore Systems

Williams et al. [WWP09] described the roofline model as a model to describe the performance limitations of computer architectures. The roofline model gives a rough estimation of the performance bottleneck behavior of a “compute kernel” or applications. The term “compute kernel” can be loosely described as code that mostly runs inside a loop. In this roofline model, a kernel is either limited by memory bandwidth or by CPU computation. The roofline model also takes the operational intensity of a kernel into account. A low operation intensity means that the ratio of bytes required to do one floating-point computation is high. One example of low operational intensity is the multiplication of sparse matrix. These matrices have a lot of zeros, which have to be transported through the memory hierarchy, however, these zeros do not increase the work of a CPU. Therefore, code with low operational intensity is more likely to be memory bound because more data has to be transported to perform an actual CPU computation. An example of high operational intensity is dense matrix multiplication. These dense matrices are filled with mostly nonzero numbers, thus, dense matrix multiplication utilizes the full CPU computation power which becomes the limiting factor. The authors also described the connection between memory hierarchy and operational intensity. A memory hierarchy with fewer cache misses will increase the operational intensity because fewer data is transferred through the whole memory hierarchy. However, the roofline model only describes the upper bound of performance and is primarily used to detect performance bottlenecks. For example, the roofline model is used to compare multiple computer architectures with each other. Furthermore, this model only considers the traffic between the caches and DRAM, e.g., the amount of data that is passed between Last Level Cache (LLC) cache and DRAM. This model does not give an exact performance prediction in the form of response time.

Ilic et al. [IPS13] extended the previously mentioned roofline model with cache awareness, e.g., it also takes memory traffic to the L1, L2, and L3 cache into account whereas the roofline model only considers the traffic to DRAM. By adding these some additional insight about bottlenecks and performance optimization can be gained. For example, some performance optimization suggestions resulting from the original roofline model are not applicable when cache awareness is integrated into the model. The roofline model with cache awareness can also find optimizations that are not observable from the original roofline model. The authors relied on hardware performance counters

to measure the cache performance and to validate their model. The roofline model with cache awareness does not give an exact performance prediction. It is rather used to gain more insight to optimize performance.

The work of Ilic et al. [IPS13] is extended with locality awareness in the Locality-Aware Roofline Model (LARM) by Denoyelle et al. [DGI+17]. In some computer architectures, the access to data is non-uniform, e.g., some computer architectures consist of multiple CPU sockets that are connected with processor-specific links (e.g., for Intel the QuickPath Interconnect). Depending on the system or code execution configuration, the required data is stored in the memory of the socket of the requesting core(local access) or the memory of another socket(remote access). This locality of data influences the memory access time. With the integration of locality awareness, Denoyelle et al. could spot additional performance bottlenecks that are not visible in the previous roofline models, e.g., the access of data with bad data locality can be limited by the remote access bandwidth. The authors provided a methodology to measure four different types of bandwidths (local, remote, contended, and congested), which provide additional upper bounds to the extended roofline model. Furthermore, the authors stated that the LARM model aims to provide insights about application characterization and optimization, and not on the exact prediction of these. The LARM model provides insightful knowledge about performance bottlenecks caused by data locality. Thus, its aim is not focused on giving an exact performance prediction.

The Execution-Cache-Model (ECM) performance model [HEF15] is an analytical performance model that can predict the amount of CPU cycles that are required to execute an application. This model similar to the cache aware roofline model takes all traffic between caches and DRAM into account. It further introduces three concepts: multicore scaling, overlap assumptions, and cache line transfer. Multicore scaling describes that some bandwidth of the memory hierarchy is scaling with increasing thread usage. The overlap assumption distinguishes if memory access to different cache levels might overlap. In his habilitation thesis, Hager [Hag, pp. 47-48] distinguishes and describes three different types of overlaps: full overlap, partial overlap, and no overlap. Full overlap means that data from L2, L3, and DRAM can be transferred simultaneously to the core. No overlap means that access to each cache level is sequential, therefore, the no overlap type is causing a longer data transfer time on the memory hierarchy. Partial overlap describes a mix between both overlap types. The ECM model also uses cache line size as the smallest unit to be transferred between the memory hierarchy to predict the performance. However, the ECM model requires the streaming assumption [Hag, p. 50] to be correct because latency effects are not modeled. The streaming assumption describes that prefetching to the cache lines should work perfectly and that data access slowdowns caused by latency are hidden [Hag, p. 32]. A further difference compared to our model is that the transfer between cache lines is measured in CPU cycles. In our approach, the actual measured bandwidth in bytes per second is used.

Xu et al. [XCDM10] proposed the CAMP model an analytical performance model that considers cache contention when two different processes run in parallel. Xu et al. defined the effective cache size of a process as the average number of cache lines occupied by the process, e.g., as a consequence two processes with high effective cache size will cause more contention, and therefore might have a slower runtime. The authors described an automated profiling process on how to extract multiple parameters with hardware performance counters for each process. These parameters have to be inserted into a non-linear equilibrium equation that must be solved to get the effective cache size for each process. With effective cache size and reuse distance of a process, the seconds per instruction of a process can be determined. With the seconds per instruction of a process information the total

runtime can be determined. However, Xu et al. only evaluated this model for two parallel running processes on a dual-core system and their work just focuses on the influence of cache contention for each process and not contention caused by memory bandwidth.

Bruns et al. [BT19] analyzed the effects of different shared resources on the performance of multicore processors. The authors conducted repeatable matrix multiplication experiments to investigate the influencing effects of shared resources. The matrix multiplication experiments are parallelized with OpenMP. All experiments are conducted on a machine with 32 physical cores. Bruns et al. analyzed five effects that can have an impact on the performance scalability with increasing threads on multicore processors:

Work distribution: This effect describes how threads are mapped to the cores. For example, there is a difference if 4 threads are distributed to 4 physical cores or to 2 physical cores with Hyper-Threading activated.

Thread count: This effect describes that for some specific number of thread usages the performance is worse. For example, Bruns et al. noticed that at uneven thread numbers small a performance degradation was observed.

Shared bandwidth: This effect describes that shared bandwidth (e.g., connections to L3-cache, connections to DRAM, or socket connections) are limited, and therefore are a boundary for scalability.

Shared storage conflict: This effect describes that some cache-misses are caused by the cache storage design. For example, the cache associativity describes how many cache lines can be stored, and shared caches such as the L3-cache will compete for these cache lines and causes additional cache misses with increasing core usage number.

Shared power and temperature: This effect describes hard physical limitations caused by power or temperature boundaries.

This study provides a good overview of different effects and their impact on multicore systems. Furthermore, their experimentation setting is similar to the setting of this thesis. Nevertheless, this study aims to understand the limiting effects of resource shared resources and did not describe a concrete measurement-based approach to predict the performance. Moreover, this study did not test the experiments with activated Hyper-Threading, e.g., it just experimented with 32 threads on 32 physical cores.

3.2 Work to Extend

In this section, the two previous works of Frank et al. [FH16] and Gruber [Gru19] are described in detail. Most ideas and concepts that are used in this thesis are based on these works. Therefore, foundational ideas and approaches of these works are described.

In an experiment conducted by Frank et al. [FH16], it was shown that Palladio has a low prediction accuracy in multicore environments (e.g., the evaluated use case has only an accuracy of 79% when the code is parallelized to use 16 cores). Frank et al. used a simple matrix multiplication application as a use case, which is modeled and simulated in Palladio. In addition, the authors executed the

Listing 3.1 Loop implementation of parallelized matrix multiplication with omp4j

```
1 // omp parallel for schedule(static) threadNum(2)
2 for (int i = 0; i < matrixA.getWidth(); i++) {
3   for (int k = 0; k < matrixB.getHeight(); k++) {
4     for (int j = 0; j < matrixA.getHeight(); j++) {
5       result[i][j] += matrixA[i][k] * matrixB[k][j];
6     }
7 }
8 }
```

matrix multiplication application on an actual multicore machine with 16 physical cores. In their experiments, the simulation time and real execution time are compared with each other to evaluate the prediction accuracy.

Different thread count variations are used to compare simulation and real execution time. In all thread count variations, two 3000x3000 matrices are multiplied with each other. The authors parallelized the matrix loop code (see Listing 3.1) with omp4j[1], which is an OpenMP like preprocessor for Java. In the first line of Listing 3.1 the threadNum(X) was varied for the thread numbers 2, 4, 8, and 16. These different thread variations are also modeled in different PCM instance variants. For example, the Service Effect Specification (SEFF) that models a matrix multiplication loop that uses 4 threads is modeled with 4 *ForkedBehaviour*s each containing an *InternalAction* with single thread CPU demand divided by 4. The authors also tested other modeling approaches, however, these approaches had some drawbacks.

Palladio is dependent on parameters such as the CPU demand and the processing rate of the modeled *ResourceContainer* on which the modeled matrix multiplication is allocated to. In their models, the processing rate is set to 1, and the CPU demand is set to the measured single thread execution time for the single thread execution case. Note that the measured execution time is only collected for the loop part of the application. Other code sections such as the setting up of both matrices with random numbers are excluded. Furthermore, the matrix multiplications were repeated multiple times, thus the mean values of all measurements were used.

In his bachelor thesis, Gruber [Gru19] addresses the low prediction accuracy on multicore environments by simulating memory hierarchy data transfers in Palladio. Gruber reused the matrix experiment that was proposed by Frank et al. [FH16]. Furthermore, he used perf to gather the amount of cache accesses to each cache level. Because Palladio does not provide model constructs to model the memory hierarchy, Gruber used Palladio's *LinkingResource*s and *ResourceContainer*s to mimic the data transfer through the memory hierarchy. The general idea is to measure the actual data amount that is transferred to each cache level and pass the data amount through *LinkingResource*s. Figure 3.1 shows an exemplary Palladio resource environment model that models caches as servers that are connected via *LinkingResource*s. Figure 3.2 shows an exemplary Palladio SEFF model that models cache accesses as *ExternalCall*s. By using an *ExternalCall* to another server, a previously measured amount of data as byte size is returned and simulated through its corresponding

[1] https://github.com/omp4j/omp4j

LinkingResource. Note that in Figure 3.2 the order of external calls does not play a role (e.g., DRAM call is before the other calls) because the actual transferred data amount between L3 and DRAM was already calculated beforehand.

Gruber evaluated his model on two different servers, one with 12 physical cores and one with 40 physical cores. Moreover, he experimented with different model variations, e.g., some model variations did not model the data transfer to the L1 or L2 cache. In his results, the modeling variations have an influence on the mean prediction error. For example, by neglecting the data transfer to L1 the mean prediction error on the 12-core server is lower, however, by neglecting the data transfer to L1 on the 40-core server the mean prediction error is higher (see [Gru19, p. 59]). Furthermore, for the 12-core server the results showed that too much simulated delay was added, e.g., for the 2, 4, 6, and 8 thread cases the predicted response time was higher than measured. For the 40-core server, the simulation missed delay. Gruber listed several limitations of his thesis, which are:

- Accesses to the memory hierarchy caused by write operations are not evaluated.
- The effects caused by additional shared caches such as L3 when multiple CPUs are used are not evaluated. For example, some machines use two or more CPUs and each of them has its own L3 cache.
- The measurements for the matrix experiment are only conducted for up to 16 threads.

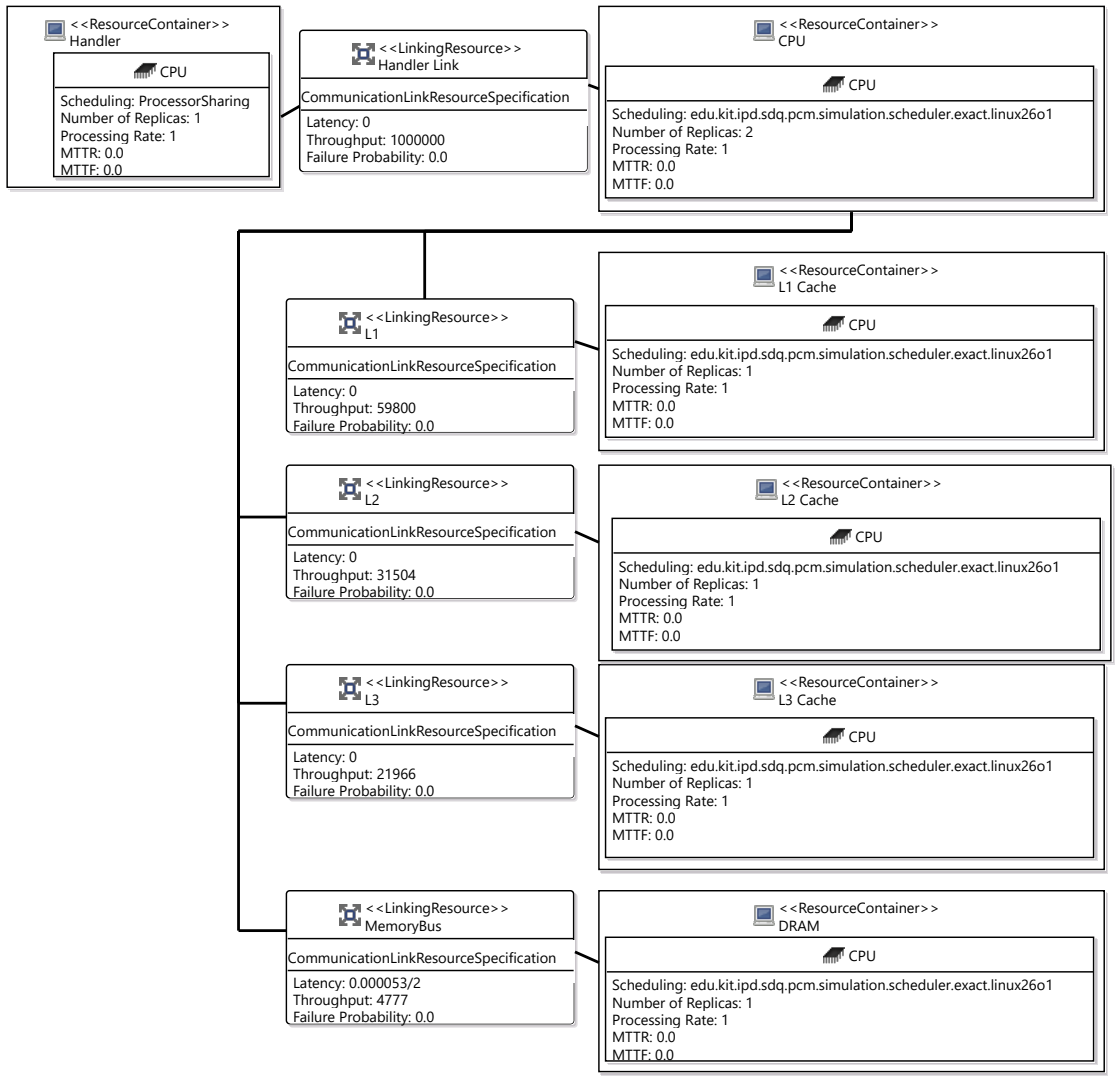


Figure 3.1: An example resource environment model from Gruber [Gru19]. The layout and naming of the original model is modified

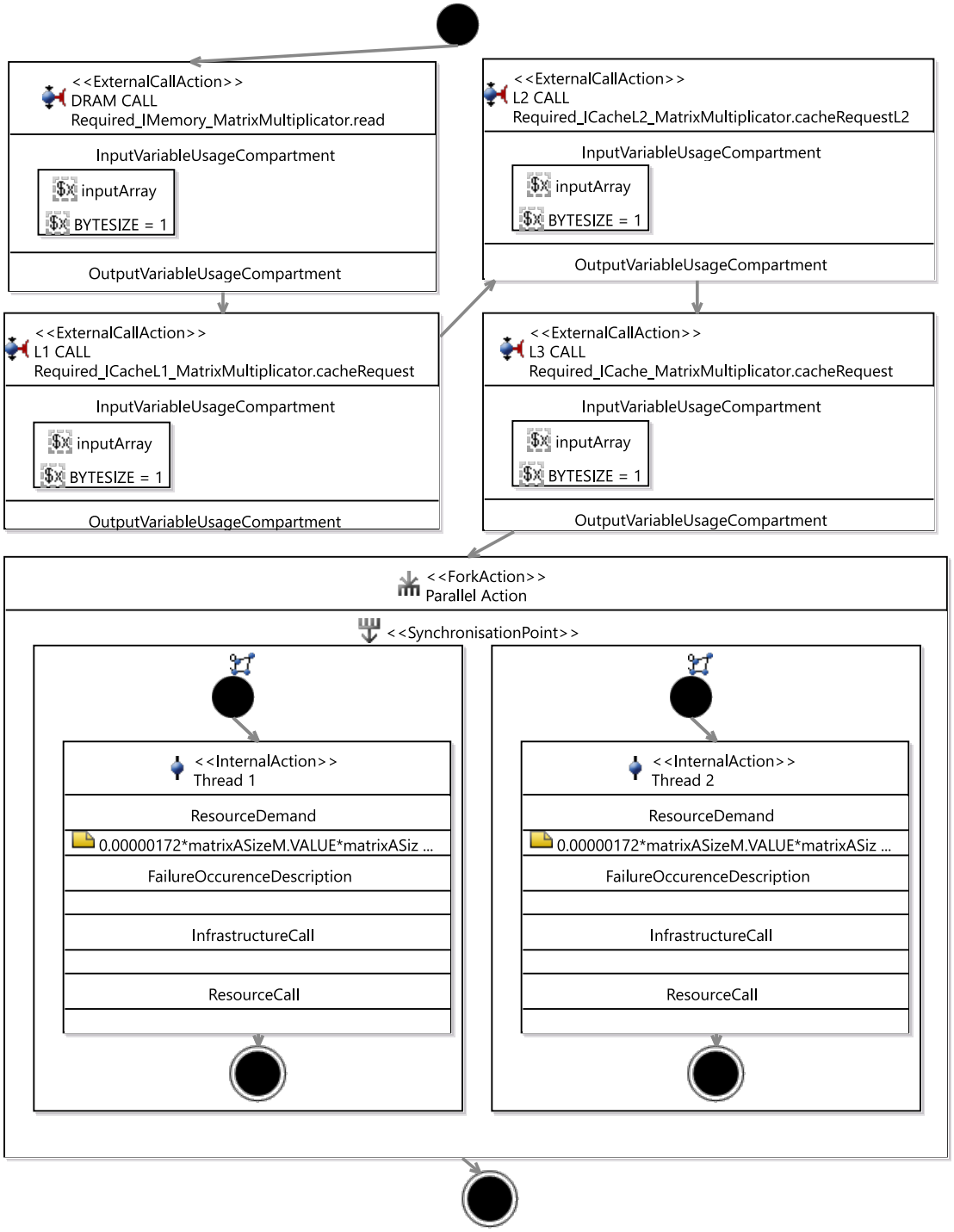


Figure 3.2: An example SEFF model from Gruber [Gru19]. The layout and naming of the original model is modified

4 Meta-Model for Memory Hierarchy

This chapter first describes why we decided to do a meta-model extension. Then, the general designing process for the initial meta-model is described, and finally, the proposed initial meta-model for the memory hierarchy is described. This chapter addresses **RQ1** *what elements are missing to model memory hierarchy and memory bandwidth* and **RQ2** *how can the PCM meta-model be extended with the identified elements*.

4.1 Possible Extension Approaches

To evaluate and model the influence of the memory hierarchy, we also checked which approach is the most suitable for our goal. In the following, we will describe which approaches were considered and why they were neglected or chosen.

4.1.1 Model Transformation

One possibility to extend Palladio models with memory hierarchy is to use model transformation. As described in Gruber [Gru19], the influence of the memory hierarchy can somehow be modeled with existing modeling elements. Therefore, a possible approach would be to check if Architectural Templates [Leh18] by Lehrig can be applied. Furthermore, Lehrig also summarized other completion approaches in [Leh18, pp. 257-261]. Approaches based on completion provide a way to specify model transformations and to pass the necessary configuration input for the model transformation. However, the configuration possibilities might be not sufficient to describe the complex structure of the memory hierarchy. Therefore, this approach was not considered further.

4.1.2 Meta-Model Extension

In this approach, the existing PCM meta-model is extended. With a meta-model extension, it is possible to define new model elements that are more suited to model the memory hierarchy. The memory hierarchy can have different designs, e.g., different amounts of cache levels. Therefore, it requires flexible model configuration possibilities. A meta-model extension allows the modeling of more complex structures and provides more modeling flexibility.

4.2 Modeling Process

This chapter describes the process we used to design an extension for the PCM to model the memory hierarchy. The goal of this process is to find a suitable meta-model of the memory hierarchy that can be used in Palladio to model and simulate the effects of the memory hierarchy.

4.2.1 Setting Goals for Modeling Process and Guidelines for Meta-model Design

As stated in the foundation chapter about the memory hierarchy in Section 2.4.2 and from the related work in Section 3.1, the memory hierarchy has a lot of properties that may have to be integrated into the model, and just to mention a few, e.g., cache size, cache associativity, cache inclusion, private or shared cache, and cache structure. It may seem intuitive that all of these properties have an impact on the performance, and therefore all should be allowed to be modeled. However, having a meta-model that allows modeling them all will increase the complexity and the time to actually model such a detailed model. In the Palladio book [RBH+16, Chapter 5] Koziolek describes how to model quality attributes such as performance, reliability, or monetary costs. He suggests a goal-driven approach with small iteration to distinguish what to model and what to omit. Koziolek described the need to focus on the following three properties:

Pragmatism: The pragmatism describes that the models should have a goal. The model design should be goal-driven and follow specific quality goals. Therefore, unnecessary work that does not contribute to the goal can be avoided. For example, in the case of this thesis, we focused only on the prediction accuracy of response time, thus, factors such as cost or reliability are not considered.

Representation: The representation describes parameters and properties that should be included in the model. A model should have all important properties that are required to achieve the goal. However, it is difficult to identify all relevant properties. Therefore, Koziolek suggests working with small iterations. In this thesis, Happe's experiment-based derivation approach, which was described in Section 1.2, is used to add new assumptions and to refine the memory hierarchy meta-model in each iteration.

Reduction: The reduction describes properties that can be simplified or ignored in the model. The reduction determines the abstraction level of the model. In general, a high abstraction level is better because the model is simpler to understand and faster to model. Unnecessary details that do not contribute to the general goal should be avoided. However, sometimes neglected properties might become important in other use cases. To support multiple use cases, Koziolek suggests either to lower the abstraction level, so that more details can be modeled, or to create multiple models.

Pragmatism, Representation, and Reduction

This section should initially address the research question **R1** to find out *what performance memory hierarchy properties that influence the performance should be modeled*? However, the final answer with all additional modeled properties that were discovered in the scope of this thesis is given in

the last Chapter 7. For the initial iteration of the meta-model, we used the approach by Koziolek as described in the previous section. Therefore, we specify the pragmatism, representation, and reduction of the model to design.

For pragmatism, we defined the goal:

Goal: Design a model that can give an accuracy gain in form of response time by including memory hierarchy factors that describe data transfer.

For representation, we initially rely on Gruber's discoveries in [Gru19] to find all elements that should be represented in the model. From the experiment of his thesis, the following structural properties and request properties are extracted:

- Structure of the memory hierarchy:
  - Cache levels, e.g., L1D, L2, L3, and DRAM.
  - Hit-rate in each cache level.
  - Data transfer links between each cache level with specific bandwidths.
  - Differentiation between shared and private caches.
- Memory requests:
  - Direct request to different cache levels,
  - Read data transfer in bytes

Gruber further listed in his threats to validity and future work section some missing properties that were not evaluated yet, which are:

- Writing data transfer
- Latency
- Synchronization effects of shared resources

For reduction, it is not simple to define which influencing factors should be neglected. The initial approach to this is that all effects that Gruber not investigated should be neglected because it is unclear how they affect the performance. However, just ignoring all these could result in bad extensibility of the meta-model, e.g., if new influencing properties are discovered and have to be integrated into the meta-model major changes might be required. In the following, neglected properties that are completely ignored in the scope of this thesis are described:

L1I cache: It might be obvious to allow the modeling of the L1 instruction cache he model at first. However, we assume that allowing to model an L1I cache does not contribute to our goal, because the performance model used by Gruber purely focuses on the memory data transfer, and we assume that the instruction transfer occurs at the same time as data transfer and is much smaller. Furthermore, in the chapter related work other, performance models did also not evaluated the influence of the L1I cache. Another disadvantage is that the L1I cache introduces parallelity (e.g., L1I and L1D access can be accessed concurrently), which might be difficult to simulate.

Cache properties abstraction : Reduction also specifies the granularity level of the model. For example, a model with higher granularity allows the modeling of additional details. In our case, it might be possible to specify caches with its cache size, cache associativity, or cache replacement strategy, which all have an impact on the hit and miss rate of the cache. Furthermore, we could model memory access patterns, and in combination with the cache properties, determine the hit and miss rates with calculations. However, we assume that calculating them like this causes imprecision, and we are still in the phase of discovering relevant performance properties of the memory hierarchy. The focus is on more precision and less on generalization. Therefore, concrete hit-rates to cache elements are used.

On the other hand, something like writing behavior or latency might have an impact on performance. Therefore, during the design of the model, we also considered these effects so that the model can be extended for these additional properties.

In summary, we want to model memory transfer through the memory hierarchy, but also keep the model extensible for a small amount. The initial goal is to get a better accuracy in form of response time by adding the memory transfer through the memory hierarchy. The initial elements to design in the model are extracted from Gruber's thesis [Gru19] because these elements seem to increase the accuracy. Some properties are completely ignored in the design phase, for example, the influence of L1I cache. More general approaches to predict or calculate the cache hit rate (e.g., with memory access pattern) are abstracted to a concrete hit rate that can be measured as demonstrated in Gruber's thesis.

Guidelines and Constraints for Meta-Model Design

In this section, a few guidelines and constraints that are followed during the design are listed. The thesis of Hauck [Hau09, pp. 21-22] listed 10 guidelines and principles that should be followed while designing a meta-model in the context of Palladio. We will summarize a few of the proposed principles by Hauck that are followed during the design of the memory hierarchy meta-model, which are:

Simplicity: The meta-model should be kept simple. Therefore, if possible, simpler abstractions should be used instead of detailed model elements. Furthermore, familiar model elements should be used if possible, e.g., by reusing existing concepts that are already in use.

Separation of roles: Specific for Palladio there are different developer roles, who are responsible to model different Palladio models. Therefore, an extended meta-model should also keep this separation. For example, the component developer, who specifies components, should not also have to model the resource environment to specify the behavior of his components.

Consistent and reasonable naming: Naming of elements should be reasonable. A name should follow its expected semantic. Therefore, the general context in which specific names are used should be respected.

Test model instances: In general the design of a meta-model should be done in small steps. Therefore, possible bugs or problems can be detected and an early stage. Thus, small prototypes should be developed in between to test for possible problems.

Furthermore, Strittmatter et al. [SHL+16] listed a few meta-model smells such as redundant container relation (e.g., avoid unnecessary association), duplicate features in sibling classes (e.g., make abstract parent classes for similar sibling classes), or dependency inversion principle violated (e.g., that classes with a high abstraction level should not be dependent on low-level classes). Therefore, these smells are avoided during the design of the memory hierarchy meta-model.

Another constraint to the meta-model extension is that it should not invasively change PCM. Therefore, no existing classes of PCM should be modified.

4.3 Meta-Model Design Rationale and Description

In this section, the design rationales during the process of designing the memory hierarchy meta-model are described. In general, we are switching between two phases: (1) identification of reusable elements in PCM and; (2) the actual design with its description. Furthermore, the following sections are divided between the design of the memory hierarchy structures and the design of a call to the memory hierarchy.

4.3.1 PCM Resource Environment Model and Reusable Elements

The benefits of reuse are: (1) familiar concepts to the modeler (e.g., the behavior in graph editor is the same) and; (2) the simulation logic of existing elements can be reused and only requires smaller adaptions.

The resource environment model specifies the hardware environment on which the software components are allocated and run. Computer nodes and linking resources are concepts that relate to the memory hierarchy. Therefore, we investigated the model elements inside the current pcm::resourceenvironment package of PCM.

The three main elements of the pcm::resourceenvironment package are:

ResourceContainer: A *ResourceContainer* represents a computer node. It can be specified with multiple *ProcessingResourceSpecification*s, which is further specified with one of the *ProcessingResourceTypes* (e.g., CPU, HDD, or Delay). These three *ProcessingResourceType*s are currently predefined in Palladio, however, it is also possible to add additional resource types. The *ProcessingResourceSpecifications* can be specified with a processing rate (e.g., CPU cycles per second in case the *ProcessingResourceType* is CPU), a scheduling policy (e.g., ProcessorSharing), and numberOfReplicas, which specifies the number of cores.

LinkingResource: A *LinkingResource* represents network links that connect computer nodes. This *LinkingResource* can be specified with a *CommunicationLinkingSpecification*, which can have different *CommunicationLinkResourceType*s. Similar to the *ProcessingResourceType*, Palladio also offers the predefined LAN *CommunicationLinkResourceType*. Furthermore, it is possible to specify throughput and latency in the *CommunicationLinkingSpecification*.

ResourceEnvironment The pcm::resourceenvironment package also offers the *ResourceEnvironment* element, which is used to store *ResourceContainer*s and *LinkingResource*s.

In the context of the memory hierarchy, the concepts of the three elements *ResourceEnvironment*, *ResourceContainer*s, and *LinkingResource*s can be modified and reused. For example, we can transfer these elements to a *MemoryHierarchyResourceEnvironment*, which stores *MemoryCache*s and *MemoryHierarchyLinkingResource*s. In the next section, the similarities and differences between the mapping of these elements are described.

4.3.2 Modeling the Memory Hierarchy Structure

Having discussed what structures are already available in the PCM, the next section addresses the model elements that should represent the memory hierarchy.

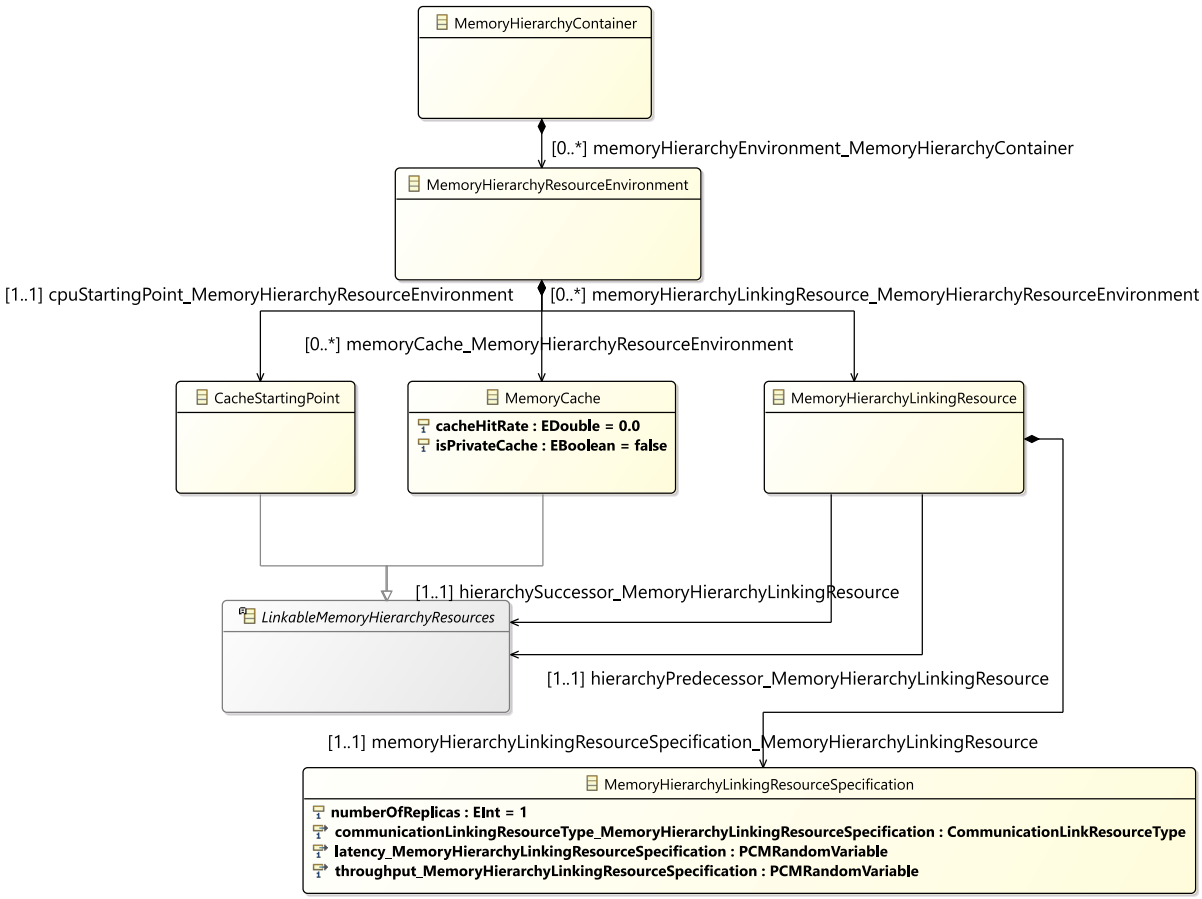


Figure 4.1: Meta-model of the memory hierarchy

In Section 4.2.1, we listed structural properties that Gruber has used to imitate the memory hierarchy structure. In the last section, we identified that we could reuse some model elements from the resource environment model and transfer it into the memory hierarchy context. With that information, we designed the memory hierarchy meta-model in Figure 4.1.

The elements Figure 4.1 are described in the following:

MemoryHierarchyContainer: It is necessary to have a root element in which all *MemoryHierarchyResourceEnvironment*s are stored. Therefore, the *MemoryHierarchyContainer* is used to act as the root element.

MemoryHierarchyResourceEnvironment: This element can contain one *CacheStartingPoint* and multiple *MemoryCache*s and *MemoryHierarchyLinkingResource*s. Therefore, each *MemoryHierarchyResourceEnvironment* contains the memory hierarchy structure for one single machine.

CacheStartingPoint: The *CacheStartingPoint* represents the entrance element of the memory hierarchy, e.g., it can represent the actual CPU cores to which the memory hierarchy is connected to. This element is also inspired by the *StartAction* from the PCM SEFF. The modeler can directly identify the start point of the memory hierarchy. Each *MemoryHierarchyResourceEnvironment* must have one *CacheStartingPoint*.

MemoryCache: This element represents a cache block, which has properties such as the isPrivate flag, which denotes if the cache is shared by multiple cores or not, and a cacheHitRate. A *MemoryCache* represents a whole cache level, e.g., if 20 L1 caches exist, then one single *MemoryCache* represents all of them. This element is similar to the *ResourceContainer* in the form that these elements can be connected via links, however, the internals is different. For example, the *MemoryCache* has no processing power in form of *ProcessingResourceSpecification*s, but it might be possible to add such a construct to model cache sharing or contention in future iterations. Currently, this element only has properties that support to model and simulate the pure data transfer.

MemoryHierarchyLinkingResource: The *MemoryHierarchyLinkingResource* represents links that connect caches. This element differs from the *LinkingResource* of PCM because it also specifies a data flow behavior. Each *MemoryHierarchyLinkingResource* has a hierarchy successor and a hierarchy predecessor. In the *LinkingResource*, the data flow is determined by an *ExternalCall*, which is defined by the component developer. In the model guidelines section, we described that a clear separation of roles is necessary. In the context of the memory hierarchy, a component developer should not know about different cache levels and only know about a start point of the memory hierarchy. Therefore, modeling the flow direction of *MemoryHierarchyLinkingResource* is added and only the system deployer should be responsible to model it.

MemoryHierarchyLinkingResourceSpecification: This element is similar to the *LinkingResourceSpecification* of PCM. Initially, we tried to reuse it, however, modeling restrictions of PCM does not allow that the *LinkingResourceSpecification* can be assigned to our *MemoryHierarchyLinkingResource*. This is because PCM defines that one *LinkingResourceSpecification* always belongs to one *LinkingResource*. As we do not plan to invasively change PCM, our own specification is created. Furthermore, it has the numberOfReplicas property, which determines how many links can be used at the same time. This can be used to model Hyper-Threading link sharing, e.g., two threads on the same physical core must share one link with each other.

The core to cache relation is abstracted in the proposed meta-model. For example, it might also be possible to model each core individually. To achieve this for a 20 core machine, 20 *CacheStartingPoint*s and 20 *MemoryCache*s representing the first cache level would be modeled. However,

this would introduce complexity. In the proposed meta-model, a few concepts are adapted from the SEFF such as starting point, or the successor and predecessor concept. Therefore, these concepts might be familiar with the modeler.

4.3.3 Integration of the Memory Hierarchy Structure into Resource Environment Model

From the current resource environment model, two elements were identified to be suitable to integrate the memory hierarchy. This section should describe how a resource container references its memory hierarchy.

The suitable elements to integrate a memory hierarchy are:

ProcessingResourceSpecification: In this case, we can just add a reference to a *ProcessingResourceSpecification* of an existing resource. As a concrete example, we could have a model instance that has a *ProcessingResourceSpecification* with CPU as *ProcessingResourceType*, and the *ProcessingResourceSpecification* references the *MemoryHierarchyResourceEnvironment*.

ResourceContainer: In this case, the memory hierarchy belongs to a *ResourceContainer*. This alternative is chosen because it is more general to assign the memory hierarchy to the resource container than to a *ProcessingResourceSpecification*.

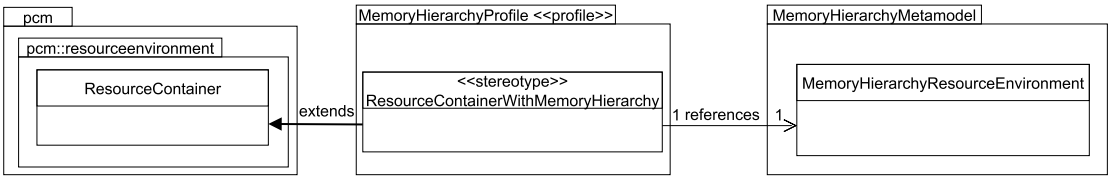


Figure 4.2: Profile and stereotype to reference a *MemoryHierarchyResourceEnvironment* to a *ResourceContainer*

As stated before, the resource container was chosen to be integrated with the memory hierarchy structure. Because we wanted to do no invasive changes to the PCM, the two alternatives were subclassing (e.g., using child extenders [Mer08], and therefore generating a subclass of the resource container with a *MemoryHierarchyResourceEnvironment* as property) or a profile-based extension [KDH+12] with profiles with stereotypes. We choose to use profiles and stereotypes to assign a *MemoryHierarchyResourceEnvironment* to a *ResourceContainer* because this approach has several advantages. An element can have multiple stereotypes at once, therefore, supporting several Palladio extensions (e.g., ResourceContainerWithPowerConsumption, ResourceContainerWithSecurity, and ResourceContainerWithMemoryHierarchy could be applied simultaneously). To achieve the same behavior with subclassing, this approach would be cumbersome (e.g., it is required to define a new ResourceContainer that inherits from ResourceContainerWithPowerConsumption, ResourceContainerWithSecurity, and ResourceContainerWithMemoryHierarchy).

Figure 4.2 describes how the stereotype approach was realized. A *ResourceContainer* of the PCM can have the «ResourceContainerWithMemoryHierarchy» stereotype. This «ResourceContainerWithMemoryHierarchy» stereotype has a tagged value that refers to a *MemoryHierarchyResourceEnvironment*. These tagged values then can be used to navigate from the *ResourceContainer* to its assigned *MemoryHierarchyResourceEnvironment*.

4.3.4 PCM Repository and Reusable Calls

Component developers require a model element that can specify how specific resources (e.g., CPU, HDD, or Delay) are used in their components. These types of model elements are calls and are specified in the pcm::seff package of PCM. For the newly introduced memory hierarchy, existing calls are evaluated to check if they can be reused. Thus, the characteristics, advantages, and disadvantages for each call are:

ExternalCall: This call is used to describe how much data and what parameters are passed from one component to another so that these parameters can be used inside the other component. The *ExternalCall* was also used in Gruber's experiment to model the influence of memory hierarchy. However, an *ExternalCall* has a reference to the *OperationRequiredRole* property, which is used by a component to determine to which other component it is connected to. This component connection relationship can not be used in the memory hierarchy context. It is possible to modify the *ExternalCall* and model an equivalent fromCache and toCache relationship, which, for example, can describe a connection between L1 and L2 cache. However, a component modeler should not know about the structure of a memory hierarchy (e.g., amount of cache levels), and therefore should not be forced to specify such a fromCache and toCache relationship. As consequence, the external action is unsuitable as a call to the memory hierarchy.

AcquireAction and ReleaseAction: In Palladio, it is possible to use these two actions to lock passive resources of a component in form of tokens. For example, these two actions could be used to limit the number of simultaneous accesses to a specific cache level (e.g., only 3 cores are allowed to access the L3 cache at the same time). However, allowing such modeling also violates the separation of roles, e.g., the component developer has to know about the structure of the memory hierarchy. Moreover, this "token" behavior is not evaluated yet, therefore, these actions are not suitable.

InternalCallAction: This call is described in the Palladio technical report [RBB+11] to allow the modeling of a nested *ResourceDemandingInternalBehaviour*, which has its own *StartAction* and *StopAction* control flow, inside the *InternalCallAction*. For example, a Java method that has several private submethods inside it can be modeled with the *InternalCallAction*. To model this example, the modeler would model the Java method as *InternalCallAction* and the submethods would be modeled as *InternalAction* inside the *ResourceDemandingInternalBehaviour*. For the memory hierarchy, this call has no benefits, thus it is not used. Furthermore, this call is currently not supported in the simulator and is also not supported in the graphical editors as described in the Jira ticket[1].

[1] https://jira.palladio-simulator.com/browse/PALLADIO-32

InternalAction: This call is used to model internal computations inside the component. It is the standard call to call resources such as CPU, HDD, or Delay. The resource demand can be specified with a *ParametricResourceDemand*, *InfrastructureCall*, or *ResourceCall*. The usage of this call in combination with a *ParametricResourceDemand* is simple. However, this call in combination with a *ParametricResourceDemand* might be not sufficient for our use case, because we want to differentiate between a writing and reading behavior. The *InfrastructureCall* and *ResourceCall* are described next.

InfrastructureCall: This call was initially conceived by Hauck [Hau09, pp. 23-36] and further introduced by Groenda in [Gro13, p. 45] as *InfrastructureCall*. This call is similar to the external call and also refers to another component. However, the targeted component by this call is a lower level component that runs on the same machine. Lower level components are software components such as the Java virtual machine or the operating system, which behaviors must be modeled inside that lower level component. For the memory hierarchy, this call can be used to call a specified component that contains the memory hierarchy logic, however, this would violate the separation of roles because the system deployer, who knows the memory hierarchy structure, must specify a component with memory hierarchy behavior. Furthermore, if the hardware deployment changes, then the repository, system, and resource environment models must be adapted. These changes might be confusing and it is hard to track all necessary changes.

ResourceCall: The resource call initially conceived in Hauck [Hau09, pp. 23-36] and further introduced by Groenda in [Gro13, pp. 44-45] as *ResourceCall* is used to call resources that provide different types of services, e.g., the HDD resource access can be differentiated as read or write, or the CPU resource can be finer described on arithmetic logic level (ALU) with MUL or ADD operations, which both may have different resource demands. The approach to define services is specified via the *ResourceRequiredRole*, which determines which resource type should be called (e.g., CPU or HDD), and the signature, which determines the service type (e.g., read or write). These two attributes must be set for the whole component before a *ResourceCall* can be used. The resource call is a suited call to be reused as a memory hierarchy call because it provides the flexibility to differentiate between service types, which can characterize the reading versus writing behavior of memory hierarchy calls.

By the investigation of the existing call actions, there are three abstraction levels:

Low: Use an *InfrastructureCall* to call a specified component with memory hierarchy logic, which can be specified via a modified external call action that can access single caches directly (e.g., for a CPU with three cache levels and main memory, four modified external calls each referring to a specific cache level would be required). This approach allows a finer modeling level and provides more flexibility because the component modeler is allowed to interact with single elements of the memory hierarchy directly and can combine them with other control flow actions of Palladio. Therefore, it might provide a higher performance accuracy. On the other hand, it violates the role separation because the component developer must have knowledge about the memory hierarchy to model this.

Medium (chosen alternative): The usage of the *ResourceCall* allows differentiating between resource call types that might be beneficial to increase the prediction accuracy. Therefore, this approach was chosen.

High: The usage of an *InternalAction* with *ParametricResourceDemand*. This approach has a high abstraction level and is simpler to understand by modelers that are unfamiliar with Palladio because only one parametric demand must be specified. The two previously mentioned approaches require the modeler to additionally model an infrastructure role or a resource role to a component, which might be difficult to understand at the beginning. However, there would be no differentiation between *ResourceCall* types. In this thesis, we assume that the differentiation might be important for our goal of higher accuracy.

Alternatively, it is also possible to model a new type of call, however, for the first initial meta-model of the memory hierarchy the *ResourceCall* in PCM seems to be sufficient.

4.3.5 Integration of Memory Hierarchy Call into SEFF Model

The *ResourceCall* already exists, therefore, no changes are required to introduce this call. However, to handle this call in the simulation, it requires invasive changes in its codebase. In our approach developing an add-on that does not require impactful changes to the code base in SimuLizar or SimuCom, we developed two variants (see Section 5.1.2). The first variant relies on code change in the SimuCom framework. If this change is integrated, then the existing *ResourceCall* can be used directly. The second variant uses an Eclipse extension point to modify the simulation behavior. However, this variant requires a new call that is defined in another package than pcm::seff. To realize this new call, Child Creation Extenders [Mer08] is used. Child Creation Extenders allows that meta-models based on EMF can be extended by other meta-models by subclassing. This extension is non-invasive, e.g., it does not change the PCM model. For the second variant, we created an *InternalActionWithMemoryHierarchy* element, which is a subclass of the *InternalAction*. Now the *ResourceCall* defined inside an *InternalActionWithMemoryHierarchy* can be interpreted in SimuLizar differently. However, the first variant is suggested, because a new *InternalActionWithMemoryHierarchy* only for the purpose of the simulation might be irritating for the modeler.

4.4 Chapter Summary

At the beginning of this chapter, we shortly compared the two approaches model transformation or meta-model extension, and decided that a meta-model extension provides more flexibility to model the memory hierarchy. Then, the goal-driven modeling process was introduced, which we used to get the initial meta-model of the memory hierarchy. Furthermore, we shortly referenced Happe's experiment-based performance model derivation process, which is followed to refine the meta-model in further iterations. Then, we described the concrete goal for the memory hierarchy meta-model of getting a better response time prediction accuracy by including memory hierarchy factors that describe data transfer. Additionally, we discussed what should be represented or neglected in the meta-model. Although we want to get the simplest model, we also bear in mind some possible extension possibilities for the meta-model for further design iterations. Before we started to design a meta-model, we looked up some guidelines, constraints, and best practices that were followed during the design process. A few sample guidelines and constraints are: the separations of roles, consistency, simplicity, and no invasive changes to PCM. Finally, we evaluated reusable concepts and elements of PCM and introduced our initial meta-model of the memory hierarchy.

5 Integrating PCM Extension in Palladio Toolchain

This chapter introduces all changes that were required to support the simulation(SimuLizar) or modeling(Sirius editor) of the proposed PCM extension.

5.1 Extend SimuLizar to Support Memory Hierarchy

The following sections should describe how we extended SimuLizar. Therefore, we address **RQ3.1** *how to adapt the simulation tool SimuLizar*. To investigate **RQ3.1**, the following sections address its subquestions, which are:

RQ3.1.1 What existing structures in PCM exist, and how do they relate to each other?

RQ3.1.2 What simulation behavior should be added?

RQ3.1.3 What must be modified to add the new behavior?

5.1.1 General Overview of Simulation Architecture

This section gives a short overview of the relationship between diverse frameworks and Palladio plugins that contribute to the simulation. It gives a broad overview of Java classes that are relevant to understand the implemented memory hierarchy extension. This section addresses **RQ3.1.1** *what existing structures in PCM exist and how do they relate to each other*.

DESMO-J/SSJ DESMO-J [GJKP13] or SSJ [LB05] are Java simulation frameworks that contain among others the implementation of an event list and the simulation clock and further classes that are required for the simulation. Palladio provides a way to select the preferred simulation engine before the start of the simulation in the Palladio Workbench [1].

[1] https://sdqweb.ipd.kit.edu/wiki/Abstract_Simulation_Engine

Abstract Simulation Engine The Abstract Simulation Engine Eclipse plugin [2] provides an abstract layer between the simulation framework of Palladio and the underlying simulation engine such as DESMO-J or SSJ. It decouples the Palladio simulation with the used simulation engine. Therefore, additional simulation engines can be added or swapped out. The Abstract Simulation Engine provides the tree main Java classes Entity, Process, and Event that interact with the event list and simulation clock of the underlying simulation engine. These three main Java classes are mapped to their corresponding classes from the DESMO-J or SSJ framework. In the following these tree classes are broadly explained:

- **Entity:** The *Entity* class represents resources (e.g., processing resources such as CPU, HDD, or network resources such as LAN)
- **Process:** The *Process* class represents more complex processes like user behavior or the behavior of forks, which describes concurrent behaviors.
- **Events:** The *Event* class is mostly used by the different schedulers to add additional events that can be used to control a finely granulated scheduling behavior.

SimuCom The SimuCom Eclipse plugin [3] provides Java classes that represent users, forked behaviors, schedulers, or resources that can interact with the underlying simulation engine. A few important Java classes are briefly described:

***ScheduledResource*:** This class represents a processing resource, such as CPU or HDD. An important method is the consumeResource() method, which defines how much demand should be simulated.

***SimulatedResourceContainer*:** This class represents the *ResourceContainer*, which can store multiple *ScheduledResource*s. An important method is the loadActiveResource() method, which determines which *ScheduledResource* should be simulated.

***SimulatedLinkingResource*:** This represents a network resource such as LAN and is similar to a *ScheduledResource*.

***SimulatedLinkingContainer*:** This class represents a *LinkingResource* and stores *SimulatedLinkingResource*. An important method is the loadActiveResource() method, which determines which *SimulatedLinkingResource* should be simulated.

***SimuComModel*:** This is the central simulation class which contains the simulation state and also providing the resource registry, which is used to store *SimulatedLinkingContainer*s and *SimulatedResourceContainer*s. The resource registry is then used during the simulation to access the container classes.

[2] https://github.com/PalladioSimulator/Palladio-Simulation-AbstractSimEngine
[3] https://github.com/PalladioSimulator/Palladio-Analyzer-SimuCom

SimuLizar The SimuLizar Eclipse plugin, [4] as described in the foundations chapter about simulation tools in Section 2.3.2, SimuLizar can additionally analyze self-adaptive systems. Furthermore, it follows an interpreted-based approach, in which the simulator traverse through a PCM instance and interprets encountered model elements. To start the interpretation, SimuLizar provides the *PCMStartInterpretationJob* class, which broadly consists of the two following steps:

1. **Create SimulizarRuntimeState:** In this step, the initial configuration of the SimuLizar analysis run is set for a *SimulizarRuntimeState* object, which provides access to all relevant objects that are necessary for the simulation. While creating a *SimulizarRuntimeState* object, different classes that have the *IModelObserver* interface can access the PCM models before the actual simulation run. For example, the *ResourceEnvironmentSyncer* has the *IModelObserver*. The *ResourceEnvironmentSyncer* is responsible to create *SimulatedResourceContainer* and *SimulatedLinkResourceContainer* for each *ResourceContainer* or *LinkingResource* that were modeled inside the resource environment model. These two containers are then stored into the resource registry of the *SimuComModel*, which is the central simulation class. This registry can be used to access these resources during the simulation. Moreover, workload drivers are initialized, which among others interprets the usage model and adds the simulated users to the event list of the underlying simulation engine.
2. **Run Simulation:** After a *SimulizarRuntimeState* object is created runSimulation() is executed to start the simulation. In the simulation, the model interpreter traverses each modeled user request and interprets them. Thereby, different Palladio models are traversed (e.g., the user behavior from the usage scenario model is interpreted, then the user's system call is interpreted, then the called repository is interpreted, then the SEFF of the repository, and so on). The interpretation of the concrete model elements is realized with switches, e.g., model elements such as an *InternalCall* have a switch-case with case:InternalCall. Inside this switch-case the logic can be implemented. For example, the *RDSeffSwitch* class contains switches for model elements defined inside the pcm::seff package. In case an *InternalCall* is interpreted, the implementation logic inside *RDSeffSwitch* would look up the resource demand and simulate this demand on the allocated resource container.

In summary, the SimuCom plugin provides among others the five classes: SimuComModel, SimulatedResourceContainer, SimulatedLinkingResourceContainer, ScheduledResource, and SimulatedLinkingResource. The container classes provides the loadActiveResource() method, and the ScheduledResource class the consumeResource() method. In a SimuLizar PCMStartInterpratation-Job are two phases: (1) creation of the different simulated containers and their resources based on the resource environment model and; (2) the simulation run which interprets for each user his chosen path defined by their calls through the Palladio models.

[4] https://github.com/PalladioSimulator/Palladio-Analyzer-SimuLizar

5.1.2 Possible Approaches to Extend SimuLizar

In the previous section, we gave an overview of the simulation relevant projects. In this section, we specify the requirements and desired behavior of the SimuLizar extension. Furthermore, we discuss different approaches to extend SimuLizar so that it can handle the memory hierarchy. This section addresses the **RQ3.1.2** *what simulation behavior should be added* and **RQ3.1.3** *what must be modified to add the new behavior*.

The following requirements should be implemented in SimuLizar:

R1 SimuLizar should be able to recognize the structure of the modeled memory hierarchy (e.g., the caches and links).

R2 SimuLizar should be able to recognize and interpret a *ResourceCall* to the memory hierarchy.

R3 The extension should have a plugin character (e.g., avoid changes to existing code if possible).

To realize **R3**, we rely on Eclipse extension points which allows Eclipse plugins to extend or customize parts of their functionality. Palladio projects provide some extension points that can be used.

To realize **R1**, on how to recognize and parse the memory hierarchy, the modelobserver extension point of SimuLizar is used.

To realize **R2**, four possibilities on how to recognize and interpret the *ResourceCall* to the memory hierarchy were identified. Before the possibilities are discussed, we describe the limitations of Palladio to customize the simulation behavior of new added *ProcessingResourceType*s next to CPU, HDD, and Delay. Furthermore, we describe the interaction sequence in SimuLizar that occur when a *ResourceCall* is recognized.

Palladio allows the creation of new *ProcessingResourceType*s inside a *ResourceRepository*. However, Palladio currently does not offer possibilities to customize the simulated consumption behavior of *ProcessingResourceType*s. Currently, only the specified processing rate, scheduling policy, and the number of replicas are used to determine the consumption behavior. If we want to specify additional factors the modeling and simulation capabilities are not sufficient.

Furthermore, we want to describe the sequence that occurs when a *ResourceCall* model element is interpreted in SimuLizar. Figure 5.1 lists a sequence of Java classes and their order in which they are called when a *ResourceCall* is interpreted. The Java classes that interact with each other in Figure 5.1 are:

***RDSeffSwitch*:** This class is used to interpret calls of the pcm::seff package. A switch-case determines which call is traversed and executes the logic for each call type.

***SimulatedResourceContainer*:** A *SimulatedResourceContainer* object represents a *ResourceContainer*. It can contain multiple *ScheduledResource*s, which represent a *ProcessingResourceSpecifcation*. Furthermore, it provides the loadActiveResource() method to initialize the simulation call of one of its *ScheduledResource*s.

***ScheduledResource*:** As a Java class representation of *ProcessingResourceSpecifcation*, this class has information about the scheduling policy or the number of replicas. Moreover, it has the consumeResource() method to initialize the *AbstractActiveResource*.

AbstractActiveResource This class represents the scheduling policy. It has a process() method that interacts with the underlying simulation engine.

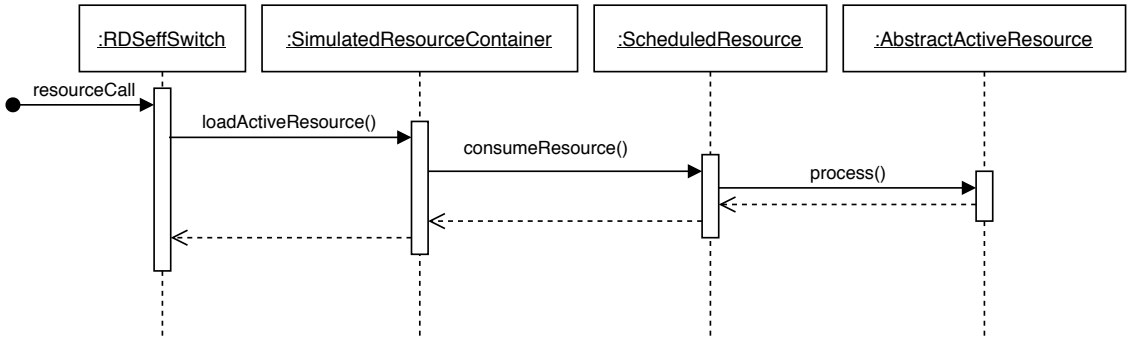


Figure 5.1: Sequence diagram in case a *ResourceCall* is interpreted in SimuLizar

After we described the simulation limitation of new *ProcessingResourceTypes* and the interaction sequence in SimuLizar when a *ResourceCall* is recognized, the four alternatives to adapt SimuLizar are discussed:

CallAction level(chosen alternative): The existing rdseff extension point can be used, which was integrated into SimuLizar in the Jira ticket[5]. This extension point does a check before a model element from the pcm::seff package is interpreted. In case the modeled element is not from the pcm::seff package, another plugin that extends the rdseff extension point is used to interpret this model element instead. For example, a switch defined in our own plugin can be used instead of the default *RDSeffSwitch*. Therefore, we can have full control over the behavior. However, this approach requires a new call action that we described in Section 4.3.5. This alternative was chosen because no major code changes in Palladio's SimuCom and SimuLizar are required. One exception is our proposed Jira ticket[6], which allows that the rdseff extension is also evaluated inside the SEFF of *ForkBehaviour*s. However, this seems to be a bug in which the extension point support for *ForkBehaviour*s was forgotten when the rdseff extension point was introduced. The rdseff extension point uses the *ComposedSwitch* of EMF. The *ComposedSwitch* "composes multiple switches to handle instances of classes defined in multiple packages"[7] .

SimulatedResourceContainer level: In this approach, the *modelobserver* extension point can be used to inject our own implementation of a *SimulatedResourceContainer* into the resource registry of the *SimuComModel* object. By having a customized *SimulatedResourceContainer*, we would have full control over its behavior. Therefore, when the *RDSeffSwitch* detects and interprets a *ResourceCall* it calls the loadActiveResource() method of the modified *SimulatedResourceContainer*. Because we have full control over the *SimulatedResourceContainer*, we can overwrite the default implementation of the loadActiveResource(). The loadActiveResource() method expects the id of the *ProcessingResourceType* as a parameter, therefore, it is possible to check if the accessed resource is a self-defined *ProcessingResourceType* and handle it. If it is another resource, then the default implementation can be used. The disadvantage

[5]https://jira.palladio-simulator.com/browse/SIMULIZAR-90
[6]https://jira.palladio-simulator.com/browse/SIMULIZAR-127
[7]https://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/util/ComposedSwitch.html

of this approach is that if other plugins would do the same and injecting their version of a *SimulatedResourceContainer*, then the behavior would be undefined because the order on how the extension points are executed determines which plugin's *SimulatedResourceContainer* version is stored or overwritten into the global resource registry of the *SimuComModel* class.

ScheduledResource (resource type) level: Currently, Palladio does not support the extension and modification of the behavior of *ScheduledResource*s with extension points. If this is allowed, then the consumeResource() behavior can be customized. For example, instead of the default behavior of calling the process() method of the *AbstractActiveResource*, which represents the scheduling policy behavior, it can instead look up the memory hierarchy and execute a customized simulation behavior. At this step, it is still possible to access the id of the associated resource container because the constructor of *ScheduledResource* requires the id of the associated resource container. The id of a resource container is required to look up its assigned *MemoryHierarchyEnvironment*. Nevertheless, this alternative requires a new extension point inside the SimuCom framework. We implemented this extension point and proposed it in the Jira ticket[8]. As described in another Jira ticket[9], the behavior of a customized consumeResource() method was also desired for the HDDProcessingResource, which simulation consumption behavior relies additionally on a specified write and read rate besides processing rate, scheduling policy, and number of replicas. The HDDProcessingResource is introduced into the SimuCom framework as the *HDDResource* Java class, which is a subclass of the *ScheduledResource*. The *HDDResource* overrides the consumeResource() method of its parent to also use a specified write and read rate in its simulation behavior. In summary, with the proposed extension point we could delegate the creation of *ScheduledResource* into external plugins and do not have to hard code them into the SimuCom framework code. Because the *ScheduledResource* is created in an external plugin, it becomes possible to give the *ScheduledResource* additional properties on creation, e.g., a registry variable that stores additional properties for its consumeResource() method.

AbstractAcitveResource (scheduling) level: Palladio allows defining new schedulers, which is described on the Palladio wiki page[10]. This approach was also considered, e.g., defining a new *ProcessingResourceType* and setting the scheduler to a self-defined scheduling policy. For example, we would define a MemoryHierarchyBus *ProcessingResourceType* and a scheduling policy with the name MemoryHierarchyBusSchedulingPolicy. As Palladio allows a customized execution of scheduling policy with its de.uka.ipd.sdq.simucomframework.resources.scheduledresource extension point, we also tried this approach. However, the customized code would be executed inside the process() method of the *AbstractActiveResource* class, and at that advanced step of the interpretation, it is not possible to access information about the *ResourceContainer*. Because of this missing information, we can not track back to the associated memory hierarchy structure.

[8] https://jira.palladio-simulator.com/browse/SIMUCOM-97
[9] https://jira.palladio-simulator.com/browse/PALLADIO-413
[10] https://sdqweb.ipd.kit.edu/wiki/Extending_PCM

5.1.3 Code Architecture

In this section, we introduce the implementation of the memory hierarchy Eclipse plugin. Firstly, we describe the main Java classes and their responsibilities in our plugin. Secondly, we describe how these classes interact with each other. Then we describe how the plugin interprets the structure of the memory hierarchy, and how it interprets the *ResourceCall*. Finally, we describe changes to existing Palladio classes, because some new introduced Java classes in the memory hierarchy plugin are adapted from exiting classes of SimuCom or SimuLizar.

Main Memory Hierarchy Plugin Classes First we introduce the main classes of the proposed memory hierarchy plugin and their responsibilities. The interaction of the introduced class can be seen in the sequence diagram 5.2 and 5.3. The main classes are:

MemoryHierarchyObserver: This class is initialized by the modelobserver extension point. Furthermore, it is responsible to parse the resource containers with a «ResourceContainerWithMemoryHierarchy» stereotype applied and pass the structure to the MemoryHierarchyRegister.

MemoryHierarchyRegister: This class is responsible to create and store the MemoryHierarchyEnvironment for each simulation.

MemoryHierarchyEnvironment: This class is responsible to create and store SimulatedMemLinkingResourceContainerWrappers. During the simulation, these objects can be retrieved. Furthermore, it also provides logic to get the successor containers.

SimulatedMemoryLinkingResourceContainer: This class represents the *MemoryHierarchyLinkingResource* This class is nearly identical to the SimulatedLinkingResourceContainer from the SimuCom framework. However, there are slight modifications, which are explained at the end of this section.

SimulatedMemLinkingResourceContainerWrapper: A wrapper class of the SimulatedMemoryLinkingResourceContainer which also has information about the hit-rate of the predecessor cache.

MemoryHierarchyCallAwareSwitch: The switch class that uses the rdseff extension point to interpret calls from other packages than pcm::seff.

As described in the previous section, the PCMStartInterpretationJob inside the SimuLizar plugin consists of two phases: (1) setup and (2) simulation.

Setup phase The interaction of our main classes in the setup phase are visualized in the sequence diagram in Figure 5.2. A part of the setup phase is that the initialize() method of all classes that use the modelobserver extension point is executed. In this phase, the *MemoryHierarchyObserver* class looks for *ResourceContainer* elements that have the «ResourceContainerWithMemoryHierarchy» stereotype applied, then looks up, creates, and stores objects representing the modeled memory hierarchy structure into a *MemoryHierarchyRegister* class. The *MemoryHierarchyRegister* stores all necessary information about the memory hierarchy structure in form of the *MemoryHierarchyEnvironment* class. Therefore, this register can be used to look up all the required memory hierarchy information during the simulation phase. Furthermore, the *MemoryHierarchyEnvironment* is also

responsible to create *SimulatedMemoryLinkingResourceContainer*, which are adapted versions of SimuCom's *SimulatedLinkingResourceContainer*, and their wrapper class *SimulatedMemLinkingResourceContainerWrapper*, which also have the hit-rate of the successor cache level. The *MemoryHierarchyEnvironment* also provides methods to return the starting point of the memory hierarchy or successors of each cache level. These methods are used during the simulation to access the successor containers.

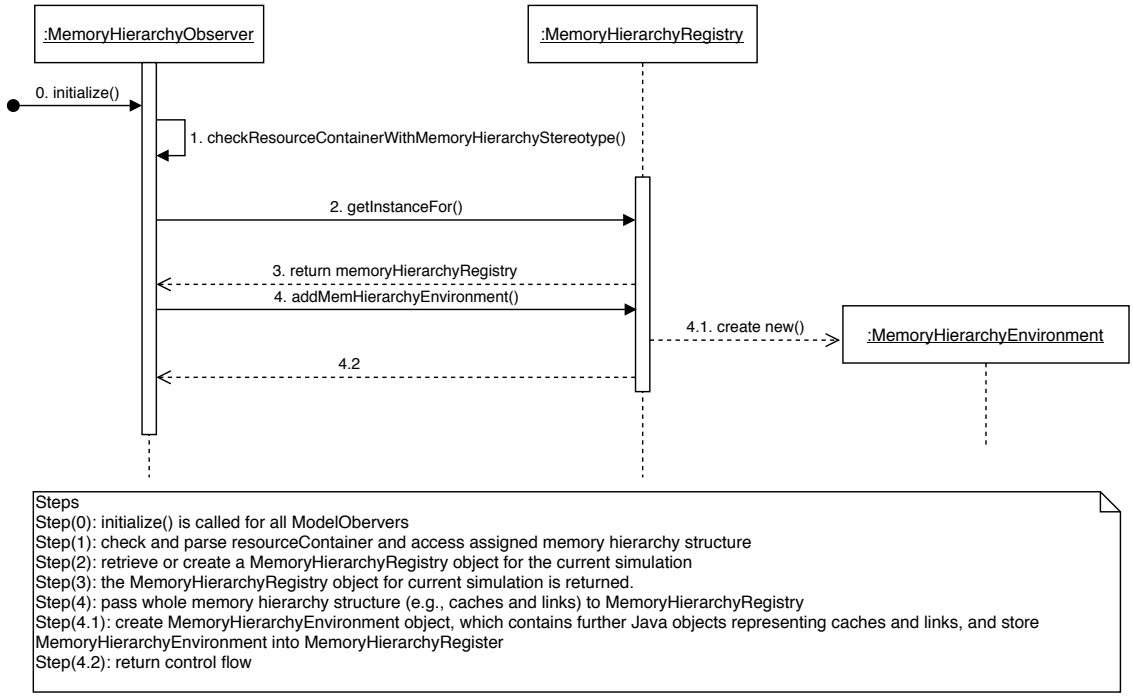


Figure 5.2: Sequence diagram for the setup phase of PCMStartInterpretationJob for memory hierarchy plugin classes

Simulation phase For the simulation phase, the memory demand is specified within the *ResourceCall* of the *InternalActionWithMemoryCall* model element, which is a subclass of the *InternalAction* and has no difference to the *InternalAction*, except that is not defined in the PCM core model, but in the memory hierarchy meta-model. Because of this, the rdseffswitch extension point can be used, which can delegate the interpretation of calls that are not specified inside the pcm::seff package of PCM to other plugins that extend this extension point. In the current implementation, the *InternalActionWithMemory* call is delegated to the *MemoryHierarchyCallAwareSwitch* of the memory hierarchy plugin. Figure 5.3 describes the general sequence during the simulation with several steps. First, the *MemoryHierarchyCallAwareSwitch* looks up the allocated *ResourceContainer* in step 1. Then, in steps 2 and 3, the *MemoryHierarchyCallAwareSwitch* passes the *SimuComModel* object and the id of the resource container to the *MemoryHierarchyRegister* to get a *MemoryHierarchyEnvironment* object that contains Java objects that describe caches and their links for the current simulation. The memory hierarchy structure is already represented as Java objects, which were created during the setup phase. Therefore, in steps 4.1 and 4.2, the initial *SimulatedMemLinkingResourceContainerWrapper* can be retrieved. The *SimulatedMemLinkingResourceContainerWrapper* represents the *MemoryHierarchyLinkingResource* element and also contains information about the hit rate of the

predecessor cache level. Thus, only the *SimulatedMemLinkingResourceContainerWrapper* object is necessary to calculate and update data demands. In step 4.3 and 4.4 of Figure 5.3, the initial memory transfer is simulated. In step 5.1 to 5.4, the successor *SimulatedMemLinkingResourceContainerWrapper*s are traversed and simulated. The calculation of the remaining demand logic is determined by the incoming memory demand and the hit-rate and the hit-rate of the predecessor cache level. For example, if 100 units were simulated through the previous link and the previous cache level has a hit-rate of 30%, then 70 units must be transferred next.

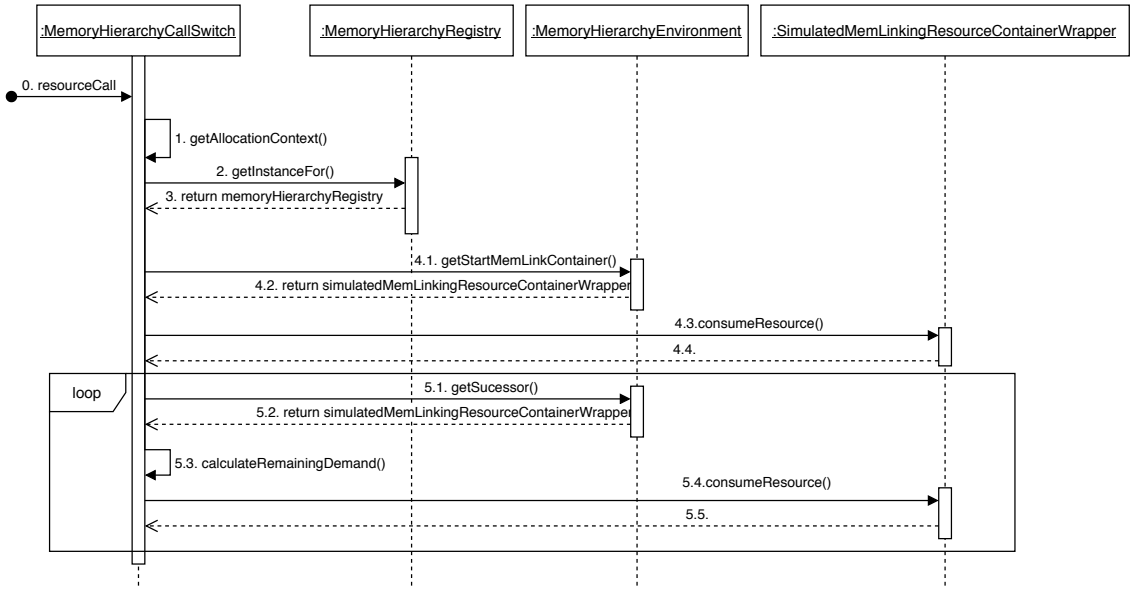


Figure 5.3: Sequence diagram for simulation sequence of memory hierarchy request when a *ResourceCall* is interpreted

In the previous Section 5.1.2, we described our proposed extension point that allows the creation of *ScheduledResource*s in external plugins. As a result, the consumeResource() method of new self-defined *ProcessingResourceType*s can be modified. If this extension point is used instead, then the steps 2 to 5 from Figure 5.3 can be moved to the modifiable consumeResource() instead.

In the following, we want to address some specific changes in our implementation of the *SimulatedLinkingResource* Java class from the SimuCom framework, which represents the *LinkingResource*. The current version of the *SimulatedLinkingResource* uses First-Come, First-Serve (FCFC) scheduling. If we just reuse this and want to model a server with 100 physical cores, which each has an own connection from the L1 to the L2 cache, then 100 of such *SimulatedLinkingResource* objects must be created in the simulation so that this private cache parallelism behavior is simulated. To avoid this overhead in the simulation, we introduced a new scheduling behavior that works similar to FCFS but allows a specific number of connections simultaneously (e.g., it processes the first n elements simultaneously instead of just one). This behavior seems to be also the desired behavior that was described in the Jira ticket[11].

[11] https://jira.palladio-simulator.com/browse/SIMUCOM-95

In contrast to SimuCom's simulation of *LinkingResource*s our version of the *MemoryHierarchyLinkingResource* does not use roundtrips. Currently, the SimuLizar does not support the simulation of *LinkingResource*s, therefore, it is not clear if SimuLizar will use roundtrips or not.

5.2 Extend Sirius Editor to Support Memory Hierarchy

Sirius-based Palladio editors provide a way to visualize and modify the different models of a PCM instance graphically. This section addresses **RQ3.2** *how to adapt the Sirius-based Palladio editors*.

We followed the guidelines and instructions on the Palladio wiki page[12] to create a new Sirius-based Palladio editor for the memory hierarchy. Therefore, we created two plugin projects org.palladiosimulator.sirius.editors.memoryhierarchy, which contains the .odesign file, and org.palladiosimulator.sirius.editors.memoryhierarchy.custom, which contains additional External Java Actions [13]. The structure of the .odesign file is showed in Figure 5.4. In general, the .odesign file has two types of elements that must be specified:

Graphical elements These elements are there to specify nodes and edges. For example, in Figure 5.4 the *MemoryCache* element in the .odesign file defines nodes representing instances of the *MemoryCache* element of the defined memory hierarchy meta-model in 4.3.2 and the *MemoryPredecessorLinkConnector* element defines edges representing the *hierarchyPredecessor_MemoryHierarchyLinkingResource* relation from the memory hierarchy meta-model.

Tools These elements are there to specify the behavior of the editor. Some tools appear in the diagram's palette editor that allows the modeler to add new nodes to the diagram (see Figure 5.5). Other tools are hidden, for example, double click behaviors on specific nodes.

The External Java Actions provide additional possibilities to modify a diagram and its underlying model with Java code. For example, an edit dialog (see Figure 5.6) can be defined with External Java Actions. The Palladio Sirius project on GitHub[14] provided a few Java classes that can be reused to implement actions such as the edit dialog.

In case the proposed extension point to define the behavior of a self-defined *ProcessingResourceType* in the simulation is accepted, the following part becomes irrelevant because the graphical modeling for the *InternalActionWithMemoryCall* is not required as it is possible to define the *ResourceCall* in the *InternalAction* of the PCM meta-model directly. For the *InternalActionWithMemoryCall*, which is a subclass of the *InternalAction*, a diagram extension[15] is used to model the *InternalActionWithMemoryCall* graphically inside Sirius-based PCM editors. In a diagram extension, a new layer is added on top of the base diagram, and on that layer new elements can be defined or existing elements can be modified. These changes are then also integrated into the base diagram and the underlying model. To view an extended diagram, the viewpoint that specifies the diagram extension must be enabled. For example, in Figure 5.7 the SeffWithMemoryHierarchy viewpoint is enabled together with the Seff viewpoint. The SeffWithMemoryHierarchy viewpoint is specified

[12] https://sdqweb.ipd.kit.edu/wiki/PCM_Development/Sirius_Editors
[13] https://www.eclipse.org/sirius/doc/specifier/general/Model_Operations.html#external_java_action
[14] https://github.com/PalladioSimulator/Palladio-Editors-Sirius
[15] https://www.eclipse.org/sirius/doc/specifier/diagrams/Diagrams.html#diagram_extension

in a .odesign file (see Figure 5.8). With the SeffWithMemoryHierarchy viewpoint enabled, it is possible to see altered graphical elements or new tools in the SEFF editor, e.g., *InternalAction*s that are *InternalActionWithMemoryCall*s have a custom green style or a new tool that creates an *InternalActionWithMemoryCall* becomes visible in the palette (see Figure 5.9).

However, this way to extend the Sirius editor has a drawback. There is a problem when multiple diagram extensions are used simultaneously. If multiple diagram extensions try to alter an element of the base diagram on their layer, then the final outcome is undefined as described on the Sirius documentation page[16].

Therefore, another extension that uses the diagram extension approach might cause conflicts with this implementation. Due to this drawback, this solution should be only temporary and the alternative approach without an *InternalActionWithMemoryCall* should be preferred. However, this disadvantage could also be hidden with a model to model transformation approach such as the Parallel Template Catalogue [FKHB19]. In this case, the modeler would use a simpler modeling construct, which is then transformed before the actual performance analysis is executed. Therefore, the *InternalActionWithMemoryCall* can be generated during the model transformation phase and must not be modeled anymore.

[16] https://www.eclipse.org/sirius/doc/specifier/diagrams/Diagrams.html#diagram_extension

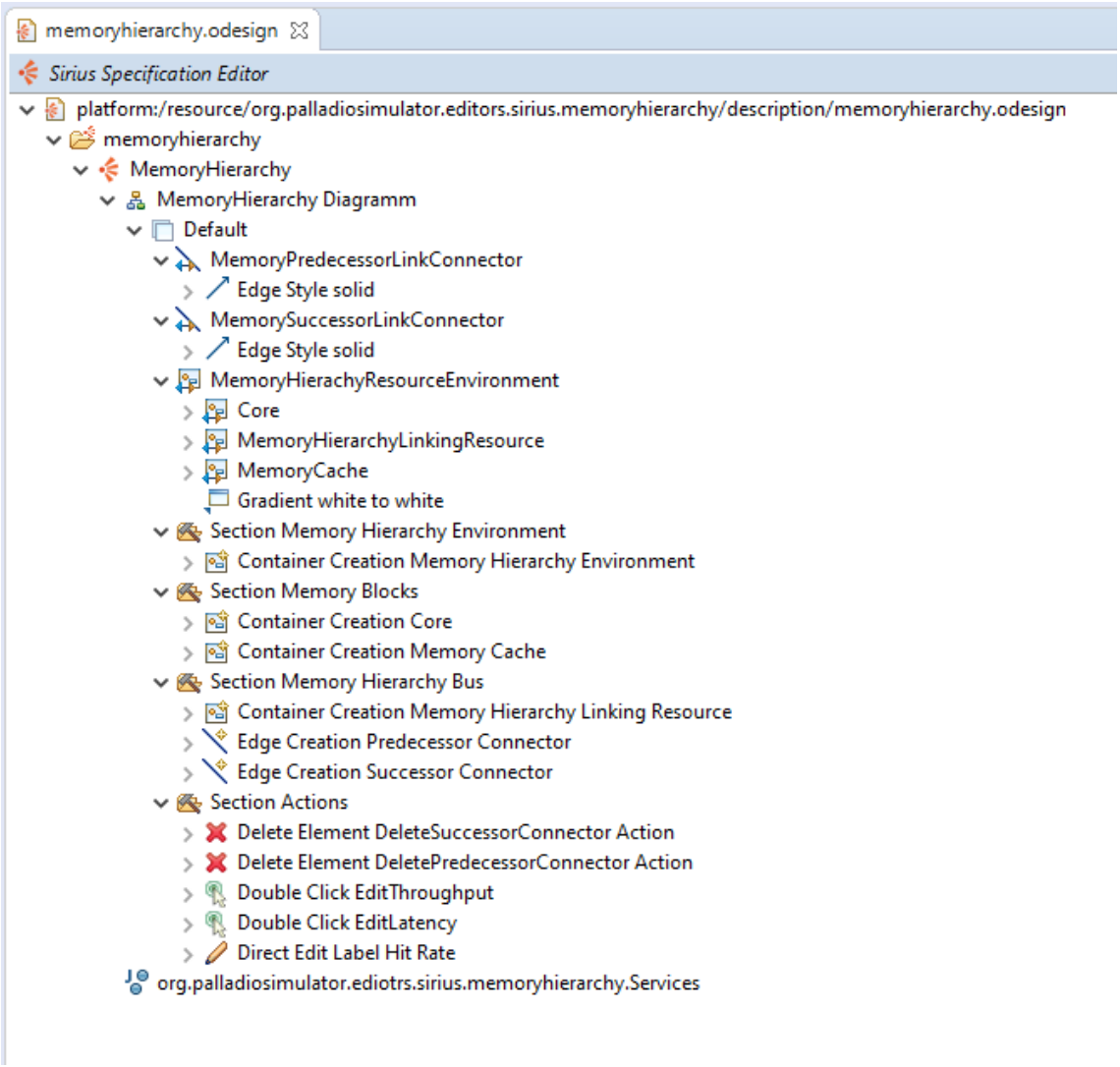


Figure 5.4: Screenshot of the .odesign file for the memory hierarchy

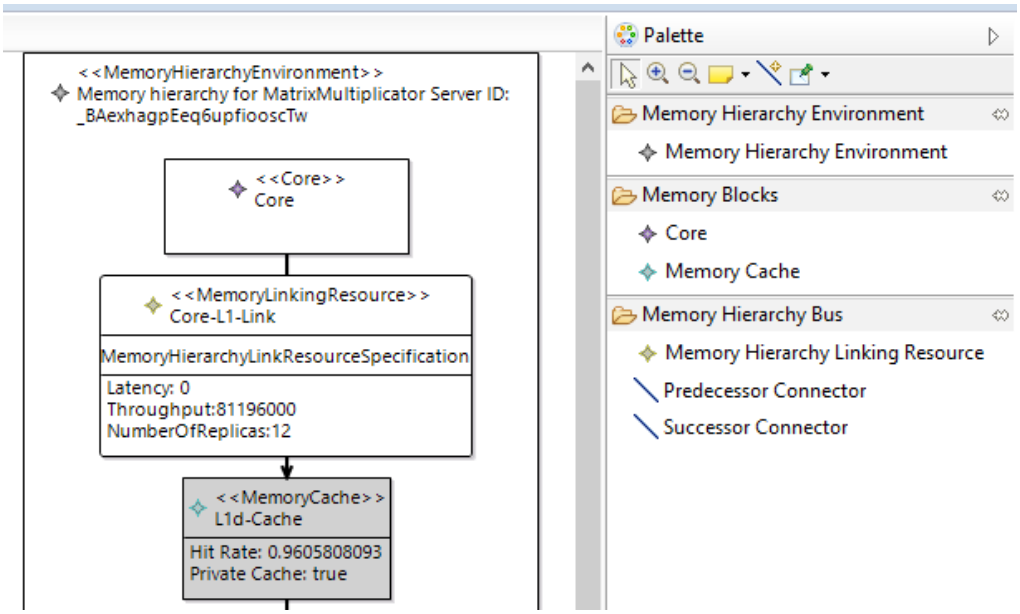


Figure 5.5: Screenshot of the memory hierarchy editor with palette showing elements that can be added to the diagram

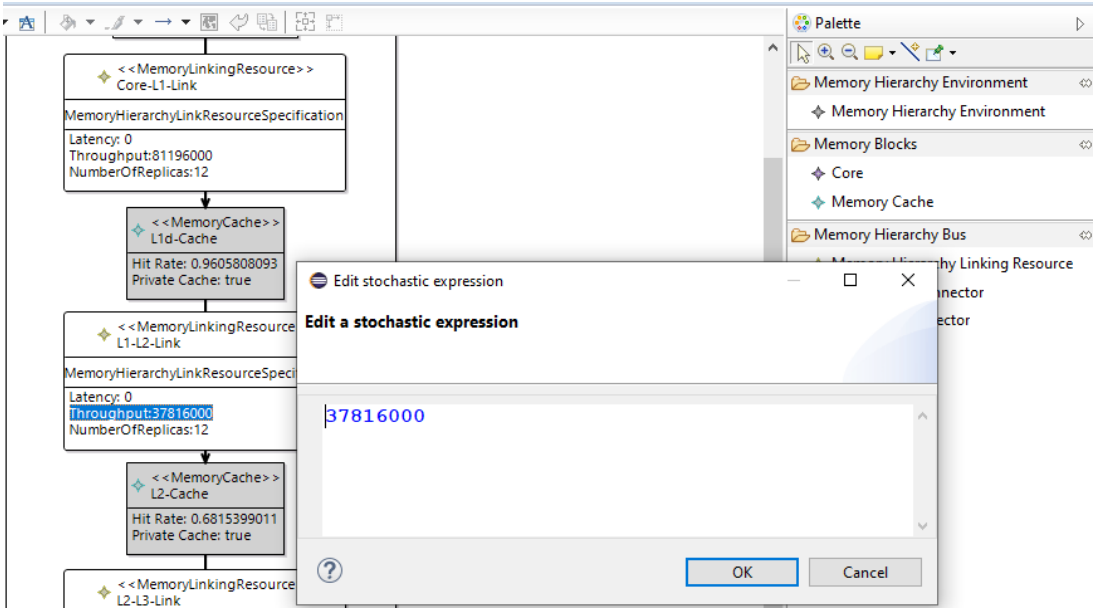


Figure 5.6: Screenshot of the memory hierarchy editor with an edit dialog

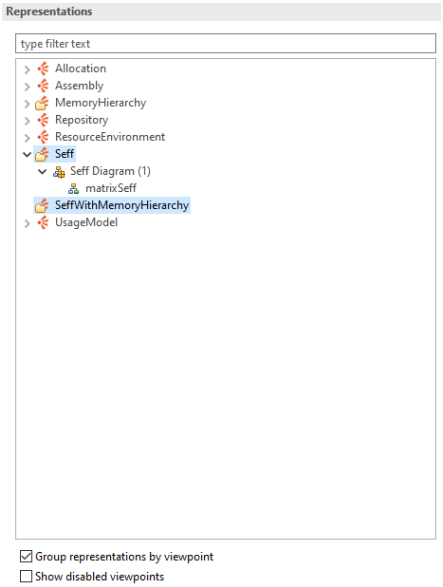


Figure 5.7: Screenshot of the Sirius viewpoint setting with the viewpoints Seff and SeffWithMemoryHierarchy activated

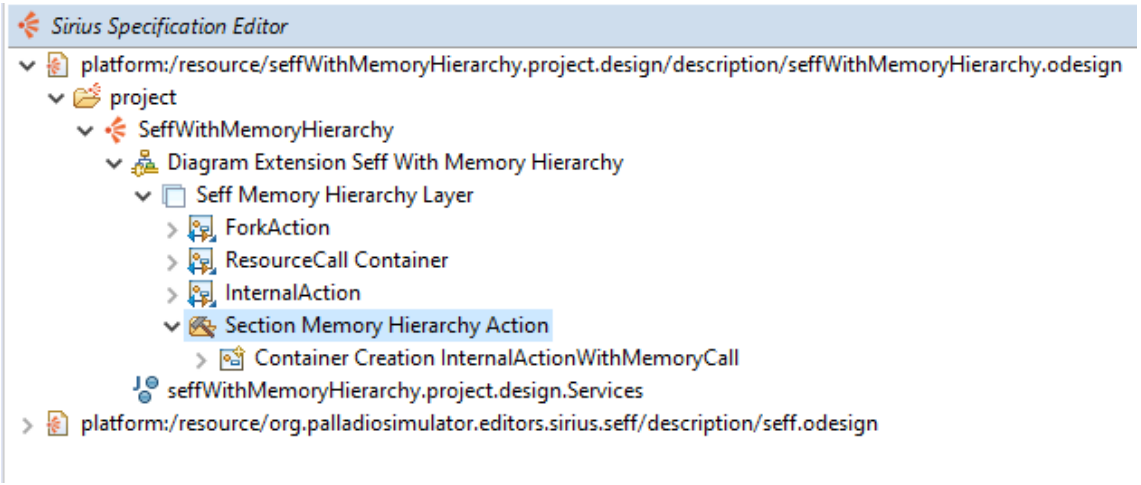


Figure 5.8: Screenshot of the .odesign file for the SeffWithMemoryHierarchy viewpoint

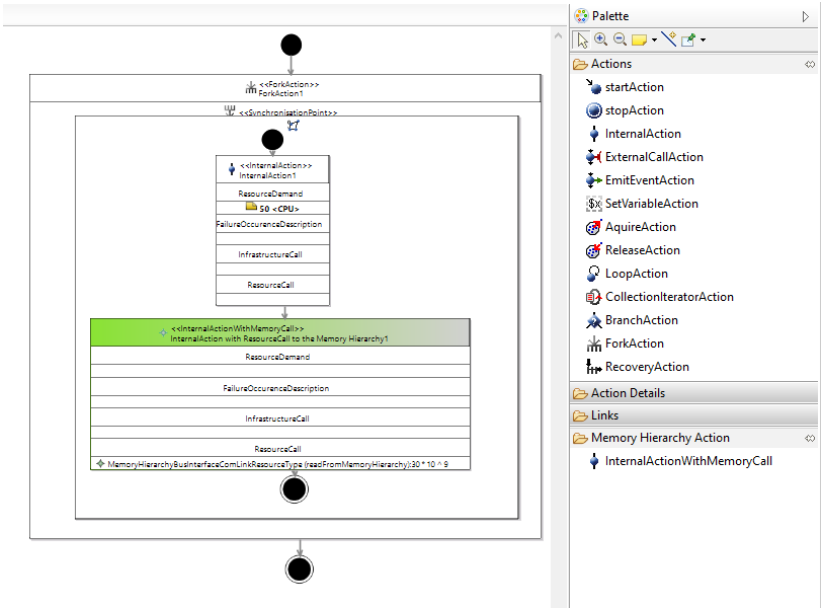


Figure 5.9: Screenshot of the editor palette with the InternalActionWithMemoryCall creation tool

5.3 Chapter Summary

At the beginning of this chapter, we gave an overview of existing plugins that contribute to the simulation in Palladio. Then, we evaluated different approaches to extend SimuLizar, and we described their advantages and disadvantages. In this thesis, we implemented two approaches. The first approach uses the existing rdseff extension point of SimuLizar. The second approach proposes a new extension point to execute custom code that specifies the simulation behavior of self-defined *ProcessingResourceType*s.

The Sirius-based Palladio editors are extended to support the graphical modeling of the memory hierarchy. To model the memory hierarchy graphically, a new MemoryHierarchy viewpoint can be selected. To model the *InternalActionWithMemoryCall* graphically, the diagram extension approach is used. The Sirius diagram extension approach for the *InternalActionWithMemoryCall* has several drawbacks. If the proposed extension point is accepted, then this call is not necessary anymore and the *ResourceCall* to a memory hierarchy resource can be directly called inside the *InternalAction* of the PCM meta-model.

6 Experimental Evaluation

This chapter describes the experiments and their results. In the first Section 6.1, we give an overview of the evaluation process. Then, in Section 6.2, we describe the experiment setting, e.g., the used machines to run the matrix multiplication experiment and the machine that was used to simulate the Palladio models. Afterward, in Section 6.3 we describe the simulated Palladio models of the matrix multiplication experiment and explain their calibration. In Section 6.4, we compare the results of the simulation with the measurements of the real experiments. In Section 6.5, we interpret the results. In the last Section 6.6, we discuss the threats to the validity of the results. This chapter addresses **RQ1** *what elements are missing to model memory hierarchy and memory bandwidth* and **RQ4** *does this extension improve the accuracy of multicore performance prediction compared to previous works*.

6.1 Evaluation Process Description

This section describes the process of how we gathered measurements and used them to calibrate Palladio models. Furthermore, the process also describes how we obtained the simulated prediction results. The whole overview of the process is presented in Figure 6.1.

Step 0: In this step, we measure the memory hierarchy bandwidths of the different cache levels with Memtest86. As a result, we get the measurements that are described in Table 6.1. Furthermore, we also measured the scaling bandwidth to the main memory, with an OpenMP parallelized version of STREAM. The results are listed in the Appendix A.4. At the same time, we modeled the matrix multiplication experiment in Palladio. The model is based on the original models from Gruber [Gru19]. However, we removed the network link behavior and integrated the memory hierarchy elements proposed in Chapter 4. A concrete example of the new model is described in Section 6.3.

Step 1: We executed the matrix multiplication experiment on three different servers. The experiment setup is described in Section 6.2. In this step, we also gathered memory cache properties with perf that are required to calculate the hit-rate of each cache level. The experiments also return the execution time for the parallelized matrix multiplication loop.

Step 2: The goal of this step is to concatenate all necessary information that is required for the model calibration into a single CSV file. Therefore, the perf memory cache and run-time measurements, which are stored as CSV files, of the previous step are processed further with python scripts, which are available on Zenodo [FT20]. For example, CPU demand units, the transferred data size through the memory hierarchy, the throughput for different cache levels, and the cache hit-rate are collected and stored into a single CSV. Finally, there is a single CSV that contains all these values for each of the three servers.

Step 3: The resulting CSVs are then parsed into a ModelBuilder [FT20] Eclipse plugin, which uses EMF factories to modify a preexisting Palladio model with Java programmatically. This plugin is required because there are cases that require modeling more than 100 *ForkedBehaviour*s. Furthermore, for each thread variation, we had to update the cache hit-rate in the memory hierarchy model. Therefore, it is not practical to model the Palladio models by hand. For this step, we also tested if the Palladio Experiment Automation[1] can be used. However, extending it so that it can access our memory hierarchy and adapt it to our use case would require too much time. Section 6.3 describes the models and their modification in detail. After each model modification, the ModelBuilder Eclipse plugin will execute a SimuLizar simulation run. The resulting simulation log with the simulated response time is stored into a txt file.

Step 4: In the last step, the response time results from the simulation log txt are processed with python scripts further to CSV files and diagrams. The outcome of this step is described in the results in Section 6.4.

[1] https://sdqweb.ipd.kit.edu/wiki/Palladio_Experiment_Automation

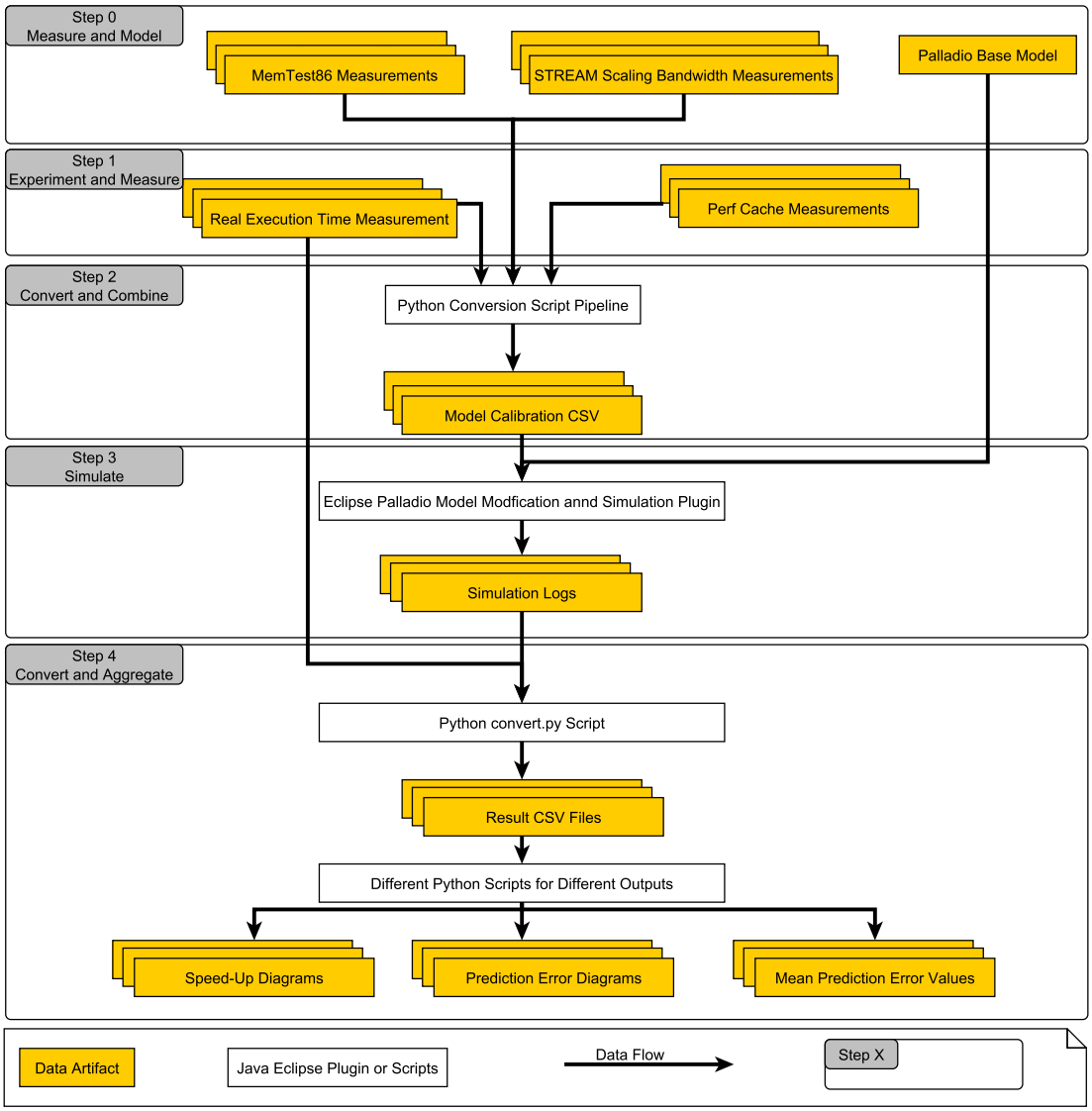


Figure 6.1: Evaluation process

6.2 Experimental Setup

The details for the three used servers are shown in Table 6.1. All three servers had Hyper-Threading enabled and were accessed via ssh. These servers are shared with other users, however, based on the experiment measurements we did not notice any interferences that are caused by other users. The cache bandwidth measurements for the 12-core server and 40-core server are reused from the MemTest86 measurements by Gruber [Gru19]. The cache bandwidth measurements for the 96-core server are also collected with MemTest86.

Processor model	Intel(R) Xeon(R) CPU E5-2640 0	Intel(R) Xeon(R) CPU E7- 4870	Intel(R) Xeon(R) Platinum 8168 CPU
Clock[GHz]	2.5	2.4	2.7
Sockets/Cores	2/6	4/10	4/24
Physical cores	2x6=12	4x10=40	4x24=96
Virtual cores	24	80	192
L1d per socket	6 x 32KiB	10 x 32KiB	24 x 32KiB
L2 per socket	6 x 256KiB	10 x 256KiB	24 x 1024KiB
L3 per socket	1 x 15MiB	1 x 30MiB	1 x 33MiB
Cache line size [Byte]	64	64	64
Measured bandwidths[GB/s]:			
Core-L1 (1 thread)	81.196	59.858	164.81
L1-L2 (1 thread)	37.816	31.504	72.59
L2-L3 (1 thread)	24.469	21.966	18.88
L3-Memory (1 thread)	7.873	4.776	8.207

Table 6.1: Server specifications and measurements

The matrix multiplication experiment was repeated 100 times on each server. Perf recorded all hardware performance counter events that are used to calculate the cache hit-rates for the 100 repeated runs at once (see Appendix A.3 for a description of the measured hardware counters and the formula to calculate the cache hit-rates). Therefore, the mean of each performance counter must be calculated to get the values for a single run. The matrix loop implementation is described in Listing 3.1. The time measurement starts before and ends after the matrix multiplication loop, e.g., matrix initialization is not included in the measurements. Furthermore, the matrix multiplication uses Pyjama [GS13], which is an OpenMP-like implementation for Java, to parallelize the multiplication loop. Pyjama allows setting how many threads should be used. This is a difference compared to [FH16] and [Gru19] because these two works used the omp4j framework[2]. The omp4j framework seems to have problems with more than 16 threads, thus, Pyjama is used. The matrix multiplication returns the run time measurement per run. To reduce the amounts of experiments that must be executed and simulated, the multiplication experiment was tested for:

- For the 40 and 96-core servers, threads between 1 and 20 in steps of 2. For the 12-core server, threads between 1 and 24 in steps of 2.

[2] https://github.com/omp4j/omp4j

- For the 40 and 96-core servers, threads between 20 and 40 in steps of 4.
- For the 40 and 96-core servers, threads between 40 and 192 in steps of 8.

In a second experiment variation, the matrix size was increased from 3000x3000 to 7000x7000 to increase the total amount of data that is transferred through the memory hierarchy. This variation leads to an increase of transferred data by a factor of 12.7, because 7000*7000*7000 loops have to be executed instead of 3000*3000*3000, therefore, $7000^3/3000^3 \approx 12.7$. In contrast to the previous experiment, this variation was only repeated 50 times for each thread variation to reduce the total experiment duration.

Jar files for each thread variation and experiment variation are created and uploaded to [FT20]. The experiments with different thread variations were executed in sequence with a bash script, e.g., execution of Jar file with 1 thread usage, then 2 threads usage, then 4 threads, and so on.

For the simulation with Palladio, the PCM nightly version between PCM 4.2.0 and PCM 4.3.0 was used. Eclipse 2019-09 Modelling tools and OpenJDK 11.0.2 were used. The Palladio simulations are executed on a Windows 10 machine with 16GB RAM and 4 x 3.2GHz.

6.3 Model Calibration

This section describes how the matrix multiplication experiment is modeled and which values were necessary to add to the model. Furthermore, this is the base model which is modified in each iteration step of the evaluation in Section 6.4.

Repository: The existing models from Frank et al. [FH16] and Gruber [Gru19] were reused and slightly modified. The model composes of an ExperimentHandler component and a MatrixMultiplicator component (see Figure 6.2). The ExperimentHandler should mimic the matrix initialization from the real executed matrix multiplication Java experiment. It is important that there is no resource demand modeled in the ExperimentHandler, because we do not want to simulate this and as described in the experiment settings Section 6.2 the initialization part is not measured in the execution. The MultiplyMatrix component models the parallelized multiplication. The SEFF of this component needs to model a *ResourceCall* to a *ResourceType*. Therefore, a *ResourceCallRole* must be specified for the MultiplyMatrix component. The *ResourceCallRole* requires a resource, which we stored inside our developed MemoryHierarchy addon[3] and can be accessed via the pathmap mechanism.

SEFF: Inside the SEFF multiple *ForkBehaviour*s must be created to represent multiple threads (see Figure 6.3). Thus, the simulation of different amounts of threads results in multiple models, e.g., a model with 2 *ForkBehaviour*s to simulate 2 threads and a model with 192 *ForkBehaviour*s to simulate 192 threads. This is error-prone and requires a lot of effort to model this by hand. Therefore, we used EMF factories to modify the models programmatically. The calibration of CPU and memory demand is described in Section 6.4 because it is dependent on the model concept with its influencing effects that was evaluated.

[3] https://github.com/PalladioSimulator/Palladio-Addon-MemoryHierarchy

ResourceEnvironment: The resource environment model consists of two *ResourceContainer*s and a *LinkingResource* (see Figure 6.4). On the *ResourceContainer* with the name *Server for ExperimentHandler* the ExperimentHandler Component is allocated. As described previously, the ExperimentHandler component does not request any resource demand. The MatrixMultiplicator component is allocated on the second *ResourceContainer*. It has a CPU as a processing resource with a processingRate of 1. The processingRate of 1 simplifies the calibration, e.g., if in the real experiment 26 seconds are measured then the requested CPU unit can be set to 26. Therefore, in combination with a processingRate of 1, the simulated time will then be approximately 26 seconds. The *ResourceContainer* with the MatrixMultiplicator allocated uses the ProcessorSharing scheduler. The *ResourceContainer* with name *Server for MatrixMultiplicator* has a stereotype «ResourceContainerWithMemoryHierarchy» applied that has a tagged value that refers to its memory hierarchy.

MemoryHierarchyModel: The *MemoryHierarchyResourceEnvironment* consist of *MemoryCache*s and *MemoryHierarchyLinkingResource*s (see Figure 6.5). The *MemoryCache*s are calibrated as follows:

Hit-rate: This is calculated based on cache accesses and misses (see Appendix A.3 for the formula to calculate this). These measured perf values are different for the different utilized thread variations. Therefore, the hit-rate for each cache must be calculated for each experimented thread variation.

isPrivate: This is set to true if we assume that the connection bandwidth to this cache scales perfectly, e.g., if 12 threads are used, then there would be 12 connections to this cache.

The *MemoryHierarchyLinkingResource*s are calibrated as follows:

throughput: The throughput was measured via MemTest86, and the results are listed in Table 6.1. The throughput values are described as bytes per ms. For example, 1GB/s would be 1 000 000 000 bytes/s, which is 1 000 000 bytes/ms

latency: This is set to 0, hence, we did not investigate the influence of latency yet.

numberOfReplicas: This is set to the amount of the actual physical core size because in the case of Hyper-Threading two virtual cores on a physical core share their connection.

We further want to mention that the CPU demand for all thread variations could also be measured per thread variation experiment. For example, the execution time of the 40 thread run can be directly used to calibrate the CPU demand model of a 40 threads Palladio model. Therefore, the simulation might be really exact. However, we want to model specifically the influence of the memory hierarchy. Therefore, we only used the execution time of the single thread run to calibrate our models. For increasing threads, the single thread run execution time is then divided by the number of threads. Therefore, there is a CPU only simulated time and a memory hierarchy only simulated time.

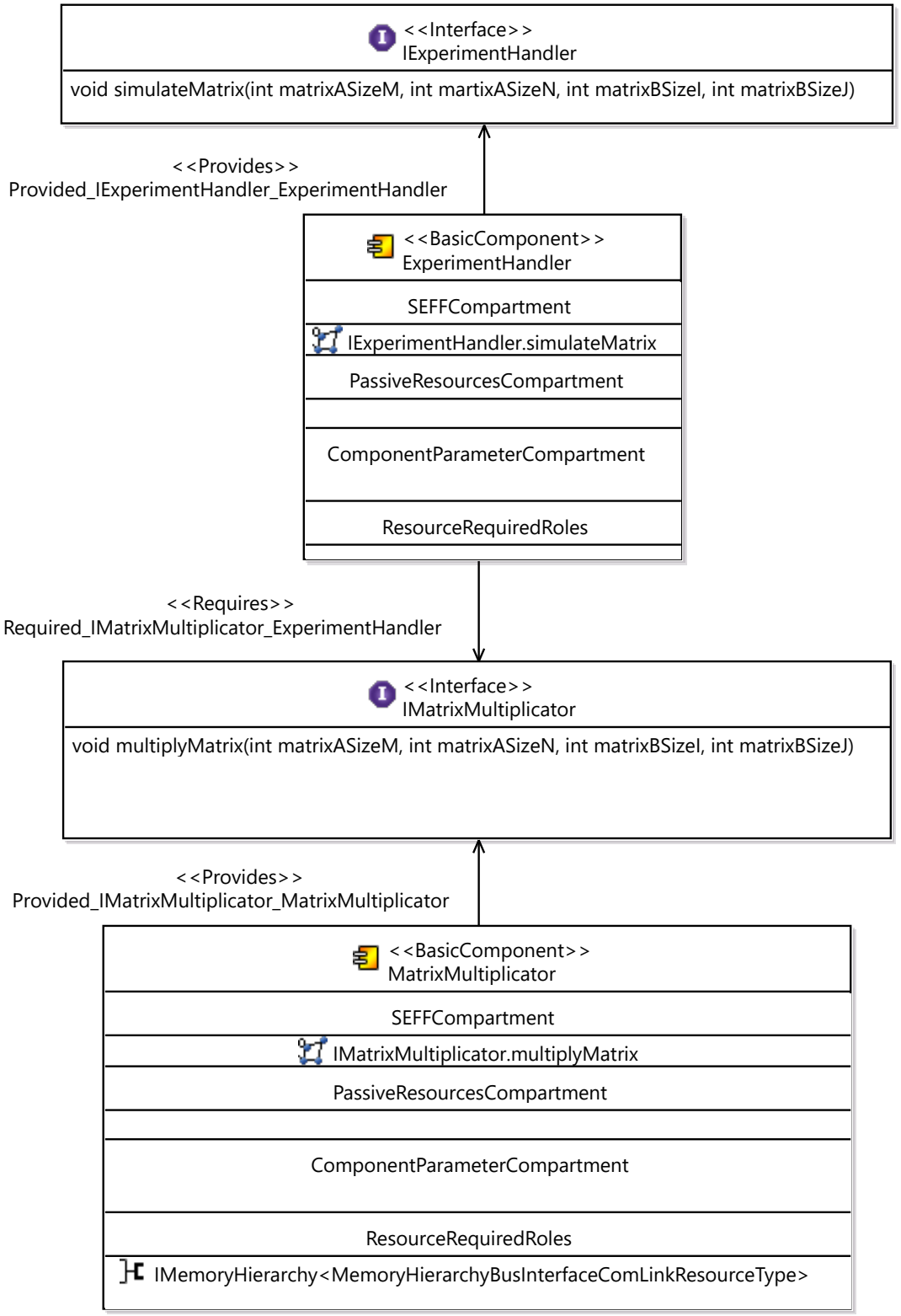


Figure 6.2: Repository model of the parallelized matrix multiplication experiment

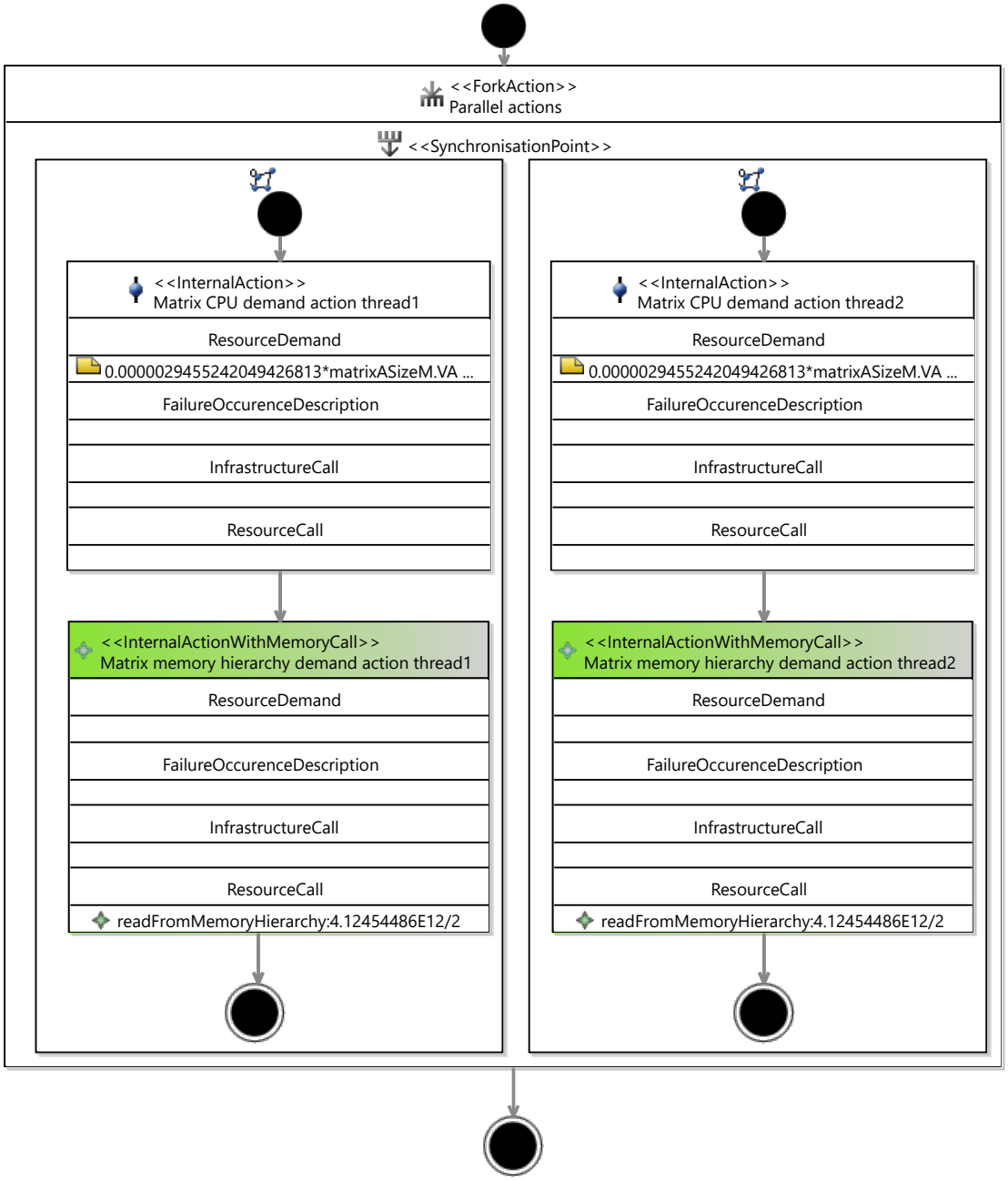


Figure 6.3: SEFF model of the parallelized matrix multiplication experiment. The *InternalAction-WithMemoryHierarchyCall* are the calls that are not inside the pcm::seff package and described in Section 4.3.5

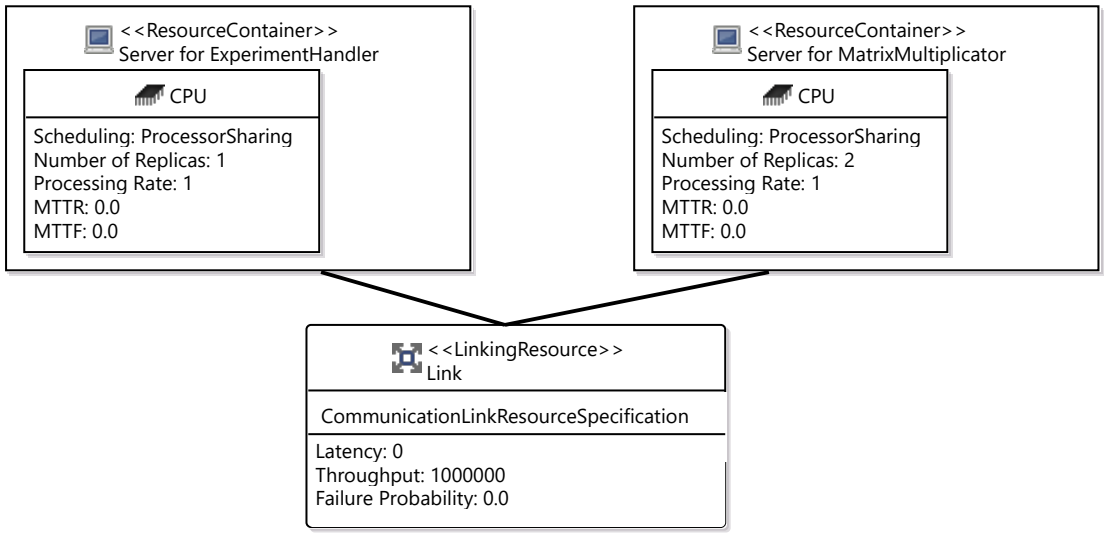


Figure 6.4: Resource environment model of the parallelized matrix multiplication experiment

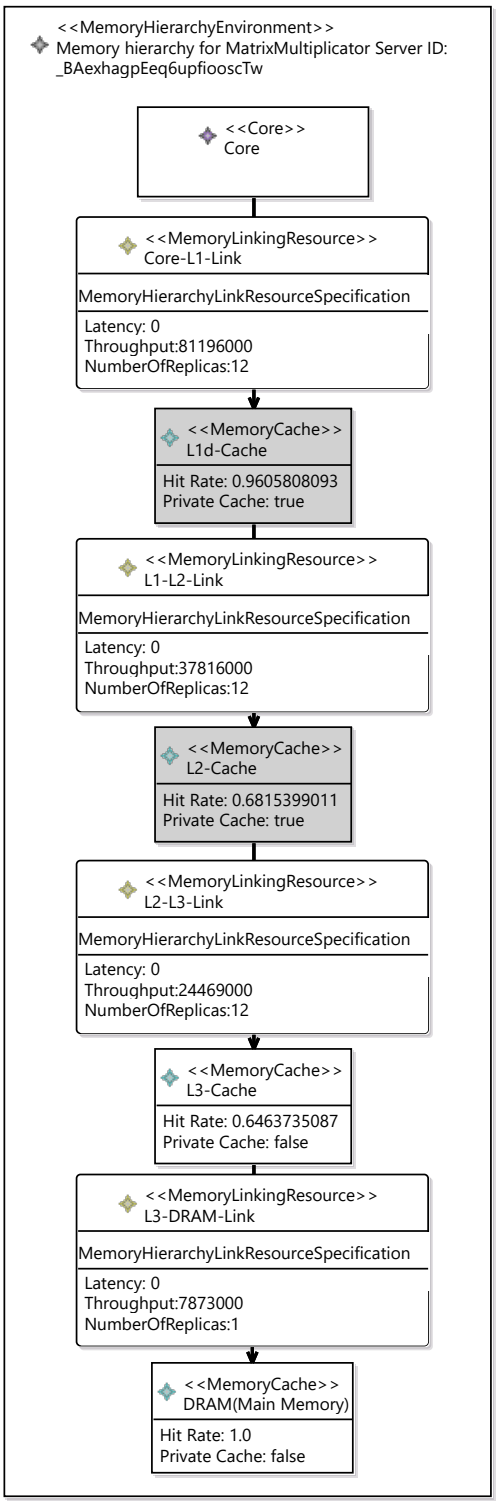


Figure 6.5: Memory hierarchy model of the parallelized matrix multiplication experiment

6.4 Results

The measurements to answer **RQ1** *what elements are missing to model memory hierarchy and memory bandwidth* and **RQ4** *does this extension improve the accuracy of multicore performance prediction compared to previous works* are described in this section. This section presents and evaluates four model iterations. For each model, necessary adaptions to the previously presented Palladio model are described. The model derivation approach is based on Happe's experiment-based performance model derivation approach (see Section 1.2). Each iteration integrated additional assumptions to improve the response time prediction accuracy. For each iteration, we validate our assumptions and evaluated the response time prediction accuracy. In each iteration, the two experiment variations with different matrix sizes are evaluated. This section refers to the two variations as exp3000 and exp7000.

For the results, we relied on the two metrics speed-up and prediction error, which are defined next. Speed-up is calculated as:

$$\text{Speed-up} = \frac{T_{sequential}}{T_{parallel}} \tag{6.1}$$

For example, if the sequential run takes 26 seconds and a parallelized run 13 seconds, then the speed-up of the parallelized run is 2.

The prediction error is calculated as:

$$\text{Prediction error} = \frac{simulated\_prediction\_time - measured\_execution\_time}{measured\_execution\_time} * 100 \tag{6.2}$$

For example, if the predicted time is 10 seconds and the measured time is 20, then the prediction error is -50%, which describes an under prediction. If the predicted time is 40 seconds and the measured time is 20, then the prediction error +50%, which describes an over prediction.

6.4.1 Observations about the Executed Experiment

Before we evaluate the different iteration steps, the speed-up characteristics of the real execution are described for the three different servers. The following results describe the speed-up behavior based on the mean time of the executed experiment (e.g., averaged over the 100 or 50 repetitions):

- 12-core server (see Experiment line in Figure 6.6): The measured speed-up peak for the 12-core server is nearly similar for both experiments at exp3000≈11.44 and exp7000≈11.03. For both experiments, the speed-up slows down after the usage of 10 threads.
- 40-core server (see Experiment line in Figure 6.7): The speed-up peak for the 40-core server is different between both experiments. In exp3000≈38 and in exp7000 ≈26. The speed-up curve on the 40-core server has fluctuations. At 14, 36, 48, and 72 threads the speed-up factor of the executed exp3000 fluctuates by a small amount and does not follow an expected trendline. Moreover, on the 40-core server the speed-up curve does not indicate that at a specific thread count the speed-up slows down. Instead, it just rises gradually. However, the speed-up is not perfect, e.g., for 80 threads the speed up in exp3000 is ≈38 and not 80.

- 96-core server (see Experiment line in Figure 6.8): Speed-up peak for the 96-core server is quite similar between experiments exp3000≈40 and exp7000≈46. At around 48 threads the speed-up slows down drastically for both experiments.

A notable observation is that in contrast to the 40-core server, the 12-core server and 96-core server have a specific point at which the speed-up slows down drastically. Furthermore, on the 96-core server, the speed-up for exp7000 is higher than then for exp3000.

6.4.2 Iteration 0

This iteration describes the previously existing model concepts from Frank et al. [FH16] and Gruber [Gru19].

Palladio-Default-Model: This is the initial model described in [FH16] in which the CPU demand of the matrix multiplication is gathered from the single thread run. For increasing thread usage the single thread demand is divided by the amount of simulated threads. For example, if 26 seconds were measured for single thread execution, then the CPU demand would be modeled as 26 CPU units and the CPU processing rate per second of the allocated *ResourceContainer* would be modeled as 1, therefore, the simulated response time would be approximately 26 seconds. If n threads are used, then n *ForkedBehaviour*s each containing one *InternalAction* with a demand of 26/n CPU units are modeled. For all simulated experiments, the CPU demand is based on the mean time of the executed experiment (e.g., averaged over the 100 or 50 repetitions)

- **Required model modifications:**
  - None
- **Results:**
  - 12-core server (see Palladio-Default line in Figure 6.6): There is a strong prediction error degradation at around 10 threads usage for both exp3000 and exp7000. In both experiment variations, the prediction error holds constants at around -10% for the usage of threads between 1 and 10. Virtual core usage starts at 14 threads, however, in both graphs the prediction error decrease already starts at 10 threads. In both prediction error graphs, the prediction error decreases steadily to approximately -50%.
  - 40-core server (see Palladio-Default line in Figure 6.7): Both prediction error lines of the two experiments have an L shape. The prediction error falls already at the usage of 4 threads for both experiments to -40%(exp3000) and -50%(exp7000). After that, in both experiment variations, the prediction error line stabilizes and decreases slowly. A further observation is the stronger fluctuation of exp3000 compared to exp7000. There are some anomalies at 14, 36, 48, and 72 threads in the exp3000 simulation. These anomalies are also visible in the speed-up curve of the executed matrix experiment.

    Furthermore, these small anomalies in the speed-up curve can result in prediction error differences of 5%. Like the other servers the prediction error line looks similar between exp3000 and exp7000, however, the prediction error from 4 threads and up differs by a large amount, e.g., at 4 threads -24%(exp3000) and -50%(exp7000).

  - 96-core server (see Palladio-Default line in Figure 6.8): For both experiments, the prediction error line can be grouped into two parts, where part one has a slower decrease than part two. For part one of exp3000, the prediction error decreases slowly from -10% to -30% between 2 and 36 threads. After that, the prediction error line decreases faster until -80%. For part one of exp7000, the prediction error decreases slowly from -10% to -30% between 2 and 48 threads. After that, the prediction error line decreases faster until it reaches -75%.

A notable observation is that on the 12-core server and 96-core server the error prediction line can be grouped into two parts. Furthermore, the 40-core server has fluctuations and a sudden prediction error drop at 4 threads. The prediction error drop at 4 threads is stronger in exp7000 than in exp3000. The fluctuations on the 40-Core server are also observable in the speed-up curve of the measured experiment.

Read-Data: The Read-Data model from Gruber [Gru19] assumes that data is transferred with a size of 4 bytes per read Java Integer. For example, in the 3000x3000 matrix multiplication 3000^3 loops are executed and in each loop iteration 3 Integers must be loaded, therefore, 320GB of data is transferred from the L1D cache. From the 320GB data the amount of cache miss is retrieved from the lower caches, e.g., if an L1D cache has 50% Hit-Rate 160GB will be retrieved from the L2 level additionally, if this L2 level has a 75% Hit-Rate 40GB will be retrieved additionally. The CPU demand is calculated like the Palladio-Default model. Compared to Gruber's original experiment, there are 2 differences: (1) The experiment in this thesis uses the proposed new modeling constructs for the memory hierarchy; and (2) the call to the memory hierarchy is inside a *ForkedBehaviour*. In Gruber's experiment, the calls are sequential, e.g., for shared links the complete memory demand to L3 is called and simulated, then the complete memory demand to DRAM is called and simulated. For example, all accesses of thread-0,..,thread-n to L3 must be finished until thread0 can access DRAM. In this case, thread-0 is dependent on thread-n and there is no parallelization. In our modeling approach, this parallelization occurs because the memory calls are inside *ForkeBehaviour*s. Therefore, modeled threads that access L3 first and finished can already access DRAM, while other threads are waiting to access L3. Because of this, the L3-DRAM transfer is faster than Gruber's experiment.

In addition, the Hyper-Threading connection limitation assumption is integrated. In the foundations chapter in Section 2.4.1 it was described that virtual cores on the same physical core share their processing resources. Therefore, we assume that the Core-L1, L1-L2, and L2-L3 connections are also limited by the amount of physical cores, e.g., each physical core has two virtual cores, however, one physical core should only provide one connection to access the cache. In this model, the maximally allowed connections to each the L1, L2 and L3 cache levels have to be set. For all model concepts, we also simulated the models without the Hyper-Threading connection limitation assumption (see Appendix A.1).

A further difference is that Gruber used LinuxO(1) as scheduling policy, which also simulates context switches [Hap09]. However, during our simulations, we noticed some problems that are described in the interpretation of results Section 6.4. Therefore, we used Processor Sharing as scheduling policy instead.

- **Required model modifications:**

  - Additional *ResourceCall* with memory demand in each *ForkBehaviour*
  - The *numberOfReplicas* value of *MemoryHierarchyLinkingResourceSpecification*s from Core-L1, L1-L2, and L2-L3 is set to the number of physical cores of each server, e.g., 12, 40, or 96.

- **Results:**
  - 12-core server (see Read line in Figure 6.6): For both experiments, the prediction error line is above the measured experiment line from 1 to 10 threads. This means that the simulated response time is longer than the measured execution time. Therefore, too much overhead is added. After 10 threads, the prediction error degrades similarly to the Palladio-Default model. In general, the shape of the Read-Data line looks similar to the Palladio-Default line.

    In the speed-up graph of both graphs, the lines are linear and reach a peak at a speed-up factor at approximately 22(exp3000) and 21(exp300) for 24 threads. The gap of the speed-up curve for exp7000 between Palladio-Default and Read-Data is larger compared to exp3000. The speed-up curve for both experiments bends slightly downwards at the beginning of 12 threads.
  - 40-core server (see Read line in Figure 6.7): The prediction error line is above the measured experiment line for the 1 and 2 threads cases in exp3000, and for the 1 thread case in exp7000. For the exp7000 the prediction error line is above the measured experiment line for the 1 thread case.

    The speed-up curve for both experiments bends slightly downwards at the beginning of 40 threads.
  - 96-core server (see Read line in Figure 6.8): The prediction error line is above the measured experiment line for the 1, 2, and 4 threads cases in exp3000, and for the 1 and 2 thread cases in exp7000. On the 96-core server, this model concept's prediction error is closest to the executed experiment from 4 to 80 threads in exp3000 and from 6 to 96 in exp7000 compared to other model concepts.

A notable observation is that for all three servers the first few threads have overpredictions. Furthermore, the speed-up curves of all three servers slightly bend when the thread number is equal to the number of cores, however, all speed-up lines are still nearly perfect linear.

6.4.3 Iteration 1

This section presents the Recalibrated-Read-Data model concept. The Read-Data model from the previous iteration has some overpredictions. For the Read-Data model concept, Gruber assumed that the overpredictions might be caused by measurement errors. However, after we reconducted the experiment and gather new measurements the overpredictions are still simulated. As a result, we assume that the modeled CPU demand might be a problem and recalibrated it. Therefore, this Recalibrated-Read-Data model concept is introduced.

Recalibrated-Read-Data: The assumption of data transfer stays the same as Read-Data. The only difference is that the CPU demand is recalculated. For this model, we separated the measured execution time into a CPU part and into a memory hierarchy part. With the example from the last model of the execution time of 26 seconds, we would also take the measured cache hit-rates from the different levels into account, e.g., the amount of time the 320GB takes through the measured bandwidth Core-L1 connection and then also the remaining data through the lower level caches and their bandwidths. The CPU part and memory hierarchy part based on perf measurements should be calculated and result to approximately 26 seconds after the calculations. A more detailed example is described in the Appendix A.2. Additionally, this model concept also uses the no overlap assumption, which was described by the ECM model (see related work Section 3.1). The no overlap assumption states that access times to all caches are sequential, e.g., the L1-L2 transfer is not in the same time as L3-DRAM transfer.

- **Required model modifications:**
  - Additional *ResourceCall* with memory demand in each *ForkBehaviour*.
  - The *numberOfReplicas* value of *MemoryHierarchyLinkingResourceSpecification*s from Core-L1, L1-L2, and L2-L3 is set to the number of physical cores of each server, e.g., 12, 40, or 96.
  - Recalibrated CPU demand in *InternalAction* in each *ForkBehaviour*.
- **Results:**
  - 12-core server (see Recalibrated-Read line Figure 6.6): In both experiments, the prediction error for the 1 thread case is 0%. In contrast to the Read-Data model concept, the Recalibrated-Read-Data model concept never overpredicts. However, from 10 threads up this model's prediction error is worse than the Read-Data model concept.

    The speed-up curve is slightly below the speed-up curve of the Read-Data model. Apart from that, the Recalibrated-Read-Data speed-up curve has the same shape as Read-Data Furthermore, its gap to Palladio-default is also higher in exp7000 than in exp3000.
  - 40-core server (see Recalibrated-Read line Figure 6.7): In both experiments, the prediction error for the 1 thread case is also at 0%. Similar to the 12-core server the Recalibrated-Read-Data model concept newer overpredicts.

    The speed-up curve is slightly below Read-Data.
  - 96-core server (see Recalibrated-Read line Figure 6.8): The prediction error has no overpredictions and speed-up behavior is similar Read-Data.

A notable observation is that for all three servers there are no overheads anymore, however, the prediction error is worse than Read-Data. Furthermore, the speed-up curves of all three servers slightly bend when the thread number is equal to the number of cores, however, all speed-up lines are still nearly perfect linear.

6.4.4 Iteration 2

This section presents the Cache-Line model concept. A reason behind this is that the previous model concept still misses overhead for increasing threads. Based on the literature review and foundations, we know that the data transfer unit between caches is a cache line.

Cache-Line: In this model, all data transfer that occurs between L1, L2, L3, and DRAM are in cache line size unit, e.g., each miss causes a load of a whole cache line of 64 bytes instead of only 4 bytes. The CPU demand is recalibrated based on this cache line assumption (see the calculation formula in the Appendix A.2). To achieve the cache-line transfer behavior in the simulation, the linking bandwidth of the L1-L2, L2-L3, and L3-DRAM connection was divided by 16. The reason behind this is that previous assumption was a 4 bytes transfer between all cache levels, now 64 bytes must be transferred through the lower cache levels, and 64 bytes is 16 times the amount of 4 bytes. Therefore, all memory links below the L1D cache level must be adapted.

The alternative requires a meta-model and simulation behavior change in which the *ResourceCall* should accept the number of instructions instead of the number of demand in bytes, because the unit of demand does not stay the same over whole memory hierarchy, e.g., the transferred demand units between core and L1D might be a Java Integer with 4 bytes and between L1D and L2 a cache-line with 64 bytes.

Furthermore, similar to the ECM model described in the related work Section 3.1, we assume that the L2-L3 connection scales like core-L1 or L1-L2 even though the L3 cache is a shared cache.

- **Required model modifications:**
  - Additional *ResourceCall* with memory demand in each *ForkBehaviour*.
  - The *numberOfReplicas* value of *MemoryHierarchyLinkingResourceSpecification*s from Core-L1, L1-L2, and L2-L3 is set to the number of physical cores of each server, e.g., 12, 40, or 96.
  - Recalibrated CPU demand for *InternalAction* in each *ForkBehaviour*
  - Throughput of *MemoryHierarchyLinkingResourceSpecification*s that transfer cache lines are divided by 16.
  - Set isPrivate value of L3 cache to false.
- **Results:**
  - 12-core server (see cache-line line in Figure 6.6): For exp3000 the prediction error is always positive with an exception to the 24 thread case. Thus, the simulated response is always slower than the measured execution time. For exp7000 simulated prediction error is always positive. The exp7000 prediction error is drastically different from the measured execution time, e.g., for some cases, the prediction error is over 100% which means that the predicted time is twice the measured time. The mean prediction error is 65.0% (see the mean prediction error Table 6.1).

The speed-up curve of exp3000 rises linear up to 10 threads, then the speed-up slows down drastically. In exp3000 from 22 to 24 threads, there is a sharp increase of the speed-up factor. For exp7000 the linear rise holds up to 10 threads before declining. The simulated speed-up curves of both experiments have a similar shape to the measured experiment.

- 40-core server (see cache-line line in Figure 6.7): The prediction error for exp3000 and exp7000 is closest to the measured experiment compared to other model concepts. For both experiments, the prediction error lines have a reverse L shape. In exp3000, the previously observed anomalies at 14, 36, 48, and 72 are observed, too. However, the anomaly at 72 threads has a bigger impact on the prediction error line of the Cache-Line model concept compared to the other model concepts. In exp7000 at 64 and up the prediction error suddenly improves.

  For both experiments, the speed-up curve of the Cache-Line model is closest to the measured experiment compared to other model concepts. In exp7000 the speed-up curve starts to sinks at 64 threads.

- 96-core server (see cache-line line in Figure 6.8): For exp3000 from 80 threads and up, the prediction error is closest to the measured experiment compared to other model concepts. For exp7000 from 96 threads and up, the prediction error is closest to the measured experiment compared to other model concepts. For both experiments, the prediction error lines have the same shape as the Palladio-Default model.

  However, in exp3000 the observed fall starts at 40 threads, therefore, later than the other models. The speed-up curves for both experiments are compared to the other model concepts the most accurate to the measured experiment.

A notable observation is that for the 40-core and 96-core server the prediction error line was closest to the measured experiment compared to the other model concepts. However, on the 12-core server the results are worse. Especially, exp7000 has significant overpredictions.

6.4.5 Iteration 3

This section presents the Cache-Line-Scaling-DRAM model concept. Based on the literature review some works indicate that the L3 connection to the main memory scales, too. Therefore, this effect is introduced in this iteration.

Cache-Line-Scaling-DRAM: In this model, we assume further that the L3-DRAM bandwidth scales with increasing core usage. However, the scaling is not perfect and limited. The bandwidth scaling is dependent on the number of used threads. For this model, the L3-DRAM bandwidth was measured for different amounts of threads. The measurement approach and measured values are listed in Appendix A.2.

- **Required model modifications:**
  - Additional *ResourceCall* with memory demand in each *ForkBehaviour*.

  - The *numberOfReplicas* value of *MemoryHierarchyLinkingResourceSpecification*s from Core-L1, L1-L2, and L2-L3 is set to the number of physical cores of each server, e.g., 12, 40, or 96.
  - Recalibrated CPU demand for *InternalAction* in each *ForkBehaviour*
  - Throughput of *MemoryHierarchyLinkingResourceSpecification*s that transfer cache lines are divided by 16
  - Set isPrivate value of L3 cache to false.
  - Throughput between L3 and DRAM is modified depending on the numbers of simulated threads.

- **Results:**
  - 12-core server (see Figure 6.6): For exp3000 at the beginning of 12 threads the prediction error line gets closer to the measured experiment than the other model concepts. However, at 24 threads it is a little worse than the 24 thread prediction error of Read-Data. For exp7000 the prediction error line overlaps with the measured experiment line at the 1, 4, 6, 8, 10, 14, and 20 thread cases. Therefore, indicating a prediction error of nearly 0% for these thread cases. The mean prediction error is 6.5 % (see Table 6.1) For the remaining thread cases, the Cache-Line-Scaling-DRAM is also the model concept with the best prediction error.

    In exp3000 the speed-up curve drops at 12 threads and indicating a slower speed-up increase similar to the measured experiment, however, from 20 to 24 threads there are some anomalies and from 22 to 24 threads there is a sharp increase of the speed-up factor. In exp7000 the speed-up curve overlaps the speed-up curve of the measured experiment for most of the threads. Exceptions in which the speed-up curve does not overlap are the threads 16, 18, 22, and 24.
  - 40-core server (see Figure 6.7): In exp300 from 1 to 40 threads the prediction error is a little better than Palladio-Default. From 48 threads and up the prediction error difference gap to Palladio-Default is greater. For both experiments, the prediction error line has a similar shape to the Palladio-Default model. The speed-up curve for exp7000 has a plateau from 72 to 80 threads, which is also observable in the speed-up curve of the measured experiment.
  - 96-core server (see Figure 6.8): In exp3000 from 1 to 64 Threads the prediction error overlaps with Palladio-Default. From 64 threads up this model concept shows a little prediction error improvement compared to the Palladio-Default model concept.

A notable observation is that for the 12-core server the prediction error line on exp7000 was closest to the measured experiment compared to the other model concepts.

		Iteration 0		Iteration 1	Iteration 2	Iteration 3
Server	Experiment Variation	Palladio-Default [%]	Read-Data [%]	Recalibrated-Read-Data [%]	Cache-Line [%]	Cache-Line-Scaling-DRAM [%]
12-core	3000x3000	27.1	16.2	20.7	**15.3**	19.6
	7000x7000	28.0	17.5	21.0	61.4	**6.5**
40-core	3000x3000	35.1	26.1	32.4	**15.8**	32.3
	7000x7000	54.3	46.9	51.8	**29.8**	50.3
96-core	3000x3000	43.9	**36.2**	41.8	37.9	41.4
	7000x7000	42.1	**34.0**	40.2	37.5	40.4

Table 6.2: Mean prediction error in percent. The smaller the error, the more accurate is the prediction. Bold values highlight the most accurate model concept for each row. The mean prediction error is averaged for thread variation described in 6.2. The mean prediction error of the 96-core server can not be used to compare with the 40-core server (see explanation in notes on results in Section 6.5).

6.5 Interpretation and Summary of Results

Notes on results Table 6.2 indicates that the mean prediction error of exp7000 on the 40-core server is higher than on the 96-core server, however, the thread steps are very big from 96 threads and up. From 96 threads and up the measurements are in steps of 8 instead of 4 as described in experiment setup Section 6.2. In both experiments, it can be seen in the Figures 6.8a and 6.8b that the prediction error from 96 threads and up is below -50%. By taking smaller steps and considering their prediction errors, the mean prediction error on the 96-core server would be worse and therefore higher.

This also influences the mean prediction error for the Read-Data model concept in Table 6.2, which shows that it has the lowest mean prediction error on the 96-core server. However, only for the first 60 threads this model concept was better than other model concepts (see Figures 6.8a and 6.8b). For higher thread cases this model concept performed worse and the thread steps from 96 and up are in steps of 8 instead of 4. Therefore, the mean prediction might be worse than the cache-line-model when measurements were done in smaller thread steps.

As mentioned in the Read-Data model concept description in 6.4.2, the LinuxO(1) was used initially as scheduling policy. However, LinuxO(1) seems to have a problem with high CPU demands, e.g., in exp7000 on the 12-core server the speed-up factor of the Palladio-Default model at 24 threads is at 24(exp7000) and 22(exp3000) and on the 40-core server the peak speed-up is at 80(exp7000) and 66(exp3000). Therefore, in the simulation for both servers of exp7000 the LinuxO(1) scheduler seems to cause no delays at all. LinuxO(1) also has problems for small response times, e.g., for simulations of the 96-core server that have response time predictions below 200ms a delay of 100ms was added on top. Therefore, the processor sharing scheduler was used instead.

Counterintuitive results The executed exp3000 has a lower speed-up peak than exp7000 on the 96-core server. For example, exp3000 has a speed-up factor of ≈40 and exp7000 has ≈46 for 196 threads. The speed-up factor on the 96-core server is only higher in exp3000 than in exp7000 for

the 2, 4, 6, 8, 10, and 36 threads cases. This result is counterintuitive because we assume that the speed-up factor on all machines for exp7000 should be worse than exp3000. The intuitive assumption is that a bigger matrix size may cause a lower speed-up factor because more data has to be transferred, and therefore the contention on the memory hierarchy should be higher. It would be interesting to find out if these measurements are faulty, or if this is a general observation.

Interesting results A small speed-up curve anomaly in the measured experiment can have an impact on the response time prediction accuracy. As observed in Figure 6.7c and described in 6.4.1 for the measured experiment, the 40-core server has fluctuations in the speed-up curve. These fluctuations of the measured experiment have an impact on the prediction error for all model concepts as shown in Figure 6.7a . Therefore, we assume that the shape of the speed-up curve might be a good visual indicator for the calibration quality of measurements (e.g., calibration of CPU demand, or the cache hit-rates). The fluctuation for the 14 and 72 threads cases can be explained with the measured DRAM accesses, which is shown in Figure 6.10a (e.g., for the 14 and 72 threads cases, the bar describing DRAM accesses is a little higher than the expected trendline).

As described in Section 6.4.4, these fluctuations have a stronger impact on the prediction error accuracy of the Cache-Line model concept, e.g., the fluctuation for the 72 thread case is stronger compared to the other model concepts as seen in Figure 6.7a. The Cache-Line model simulates more data through the L3 and DRAM connection than other model concepts, therefore, the previously observed DRAM access increase for 72 threads influences the cache-line model concept stronger than other concepts.

The prediction error for simulated models gets worse gradually with increasing thread usage, however, there are some outliers. Some outliers can be explained with the measured cache accesses between L3 and DRAM. For example, in exp7000 on the 12-core server the prediction error for the 16 threads case of all model concepts do not follow the trendline (e.g., for models with cache line, there is a peak and for the other models there is a small plateau between thread variations 14 and 16 for the prediction error as it is seen in Figure 6.6b). This can be explained with higher DRAM cache accesses for the 16 threads case, which is shown in Figure 6.9b. Similarly, in exp7000 on the 40-core server the outliers for the 72 and 80 threads cases can be observed (see the two cache-line model concepts in Figure 6.7b). This can be explained with a higher DRAM cache access in the 72, and 80 threads cases (see Figure 6.10b).

Sudden prediction error drop on the 40-core server from 4 threads to 6 threads in exp3000 and exp7000 for some model concepts can be explained with the sudden lowering of DRAM accesses between 4 and 8 threads (see both subfigures in Figure 6.10)

Predication error difference between exp3000 and exp7000 on 40-core server can be explained by the speed-up curves 6.7c and 6.7d. The speed-up curve increases slower in exp3000 than in exp 7000. A known observation is that more linearity in the speed-up curve (e.g., if the speed-up factor is nearly be as high as the used thread amount) results in less prediction error. Therefore, the flatter speed-up curve of the measured exp7000 compared to exp3000 causes the higher prediction error in exp7000 on the 40-core server. Similarly, the prediction error plateau from 8 to 32 threads on the 96-core server for all simulated models can be explained with the stepper speed-up curve from 1 to 32 threads which are close to the measured speed-up curve (see the Figures 6.8c and 6.8c).

There is no influence of Hyper-Threading-connection limitation on the Cache-Line model and only a small influence on Cache-Line-Scaling-DRAM model as listed in the Appendix in Table A.1 or Figures A.1, A.2, and A.3. The cause for this is that we modeled the connection from L3 to DRAM as a single connection and all threads have to wait for their turn to use this connection (e.g., the usage is sequential). In both models involving the cache line, this single connection becomes the bottleneck in the simulation. This bottleneck outweighs the slowdown caused by the connection limit of the upper memory hierarchy. For example, if the 96-core server has a limit of 96 connections between Core-L1D, L1-L2, and L2-L3 and the usage of 192 threads is modeled and simulated, then half of the 192 threads (e.g., t97,t98,...,t191, and t192) must wait on each cache level, however, the models with cache line are still so slow that the threads that are allowed to access the DRAM first (e.g., t1,t2,...,t95, and, t96) are not finished when the slow downed threads want to access DRAM.

Expected results As observed in Table 6.2 and Figure 6.6b the mean prediction error is the lowest for exp7000 on the 12-core server for the Cache-Scaling-DRAM model. The mean prediction error for the Cache-Scaling-DRAM model is remarkable because the mean prediction error for this model approach is 6.5% and thus much lower than other results in Table 6.2. As described in the related work Chapter 3.1 the ECM model used cache-line transfer, and scaling L2-L3 and scaling L3-DRAM connections. The ECM model is based on the assumption that L3-DRAM is a potential bottleneck. If we force this bottleneck then the performance model should have an impact on the prediction and should be more accurate. We assumed that if the bandwidth is saturated and reaches its limit, then a saturated bandwidth effect between L3-DRAM should be clearly measurable. This was also the primary reason why the experiment was scaled higher because from the theoretical calculation of the perf measurements we assumed that in the 24 thread case a total of approximately 388GB would be transferred through L3-DRAM, which would saturate this connection with a scaled L3-DRAM bandwidth of approximately 18.36GB for 24 threads (see STREAM benchmark measurements of 12-core server in Appendix A.4). In our data in [FT20], we measured DRAM access of 6074849580 times in the 24 threads case on the 12-core server. If this is multiplied with 64 bytes, which represents a cache line transfer, then 388GB is transferred. Furthermore, the integration of scaling of L3-DRAM and L2-L3 seems reasonable because else the model would overpredict by a large amount as it is seen for the cache-line model concept in Figure 6.6b for exp7000 on the 12-core server

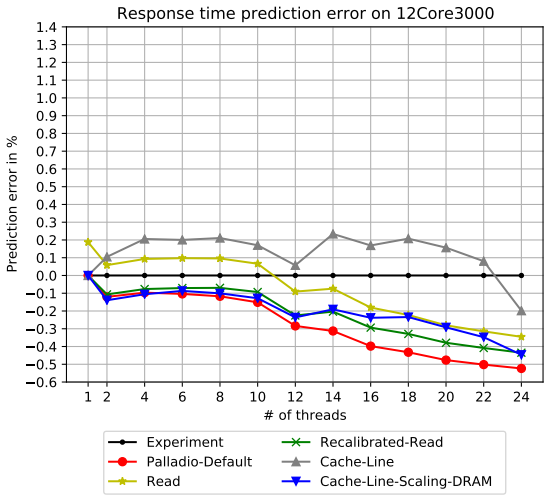
Unexplainable results and assumptions For the 96-core server there seems to be a clear bottleneck that is not modeled in the current models. The bottleneck seems to be at approximately 48 threads. This thread number might be not random. As the 96-core server has 4 CPUs each with 24 cores, we assume that either two CPU sockets each with 24 cores are utilized or one socket with 24 cores and 48 virtual cores are used. Therefore, an assumption might be that the socket interconnects could have an influence on the performance.

On the 96-core server, we saw a specific pattern in the cache access measurements regarding the L1 cache access. Figure 6.12 shows that starting from 96 threads on the access to L1D rises by a significant amount for exp3000 and exp7000. This might be an effect that is caused by Hyper-Threading (e.g., from 96 threads an up virtual-cores are used in addition). However, these findings can not be found in the measured execution time of the experiments and the influence of them are unclear.

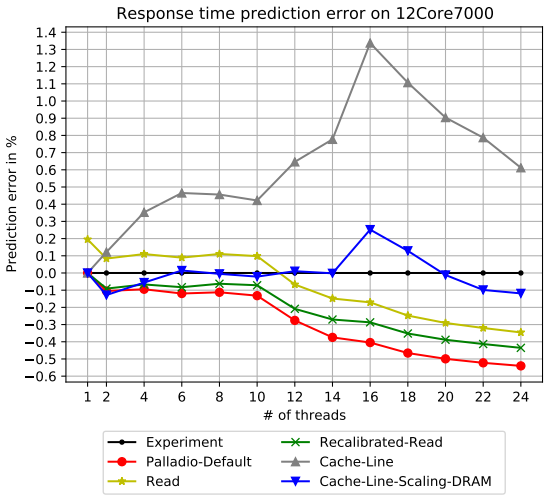
A further anomaly observed on the 96-core server was that the DRAM access for exp7000 was approximately 23 times higher than exp3000 for the 1 thread case (see caption of Figure 6.11). On the 12-core server and 40-core server, the difference between exp7000 and exp3000 is approximately 12, which seems to be reasonable because exp7000 is scaled by approximately factor 12 as described in the experimental setup Section 6.2. Therefore, multiple surprising measurements on the 96-core are observed after inspecting the results closer.

Results summary In the following, the five model concepts are summarized. The Palladio-Default model concept does not simulate the memory hierarchy, and the speed-up is perfectly linear. The Read-Data model concept adds overhead because the model uses a higher base CPU demand. However, this model has some over predictions. The Recalibrated-Read-Data model concept does not have any overpredictions but it performs worse than the Read-Data model concept. The Cache-Line model concept performed well on multiple servers because overhead was missing and this model concept adds the most overhead. However, as seen on the 12-core server with exp7000 it has the worst mean error prediction of 65%. This worse mean error prediction is caused by over prediction. Finally, Cache-Line-Scaling-DRAM model concept performed best with a mean prediction error of 6.5% when the L3 to DRAM connection is saturated. However, in other cases, this model concept did not add enough delay. Furthermore, the Hyper-Threading connection assumption has a very small effect on model concepts with cache lines.

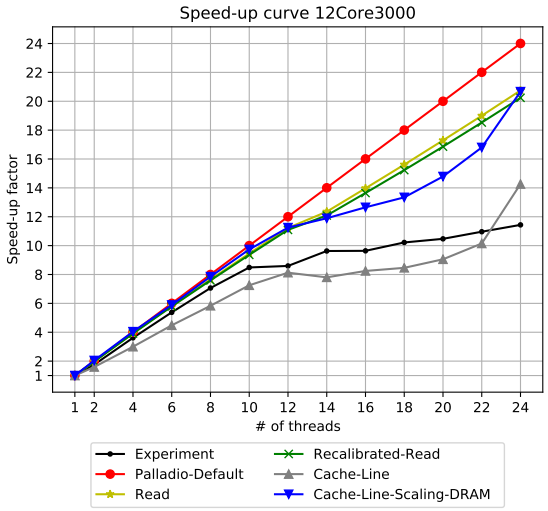
Most observed outliers in prediction error can be explained by the speed-up curves of the measured experiments. Therefore, fluctuations in the measured experiment influence the simulated prediction accuracy. A few outliers can also be explained further by the measured DRAM access rate. There are two factors that keep the prediction errors low. The first factor is that when the measured speed-up is linear. The second factor is that when there is high measured access to DRAM because this adds more delay to the simulation.



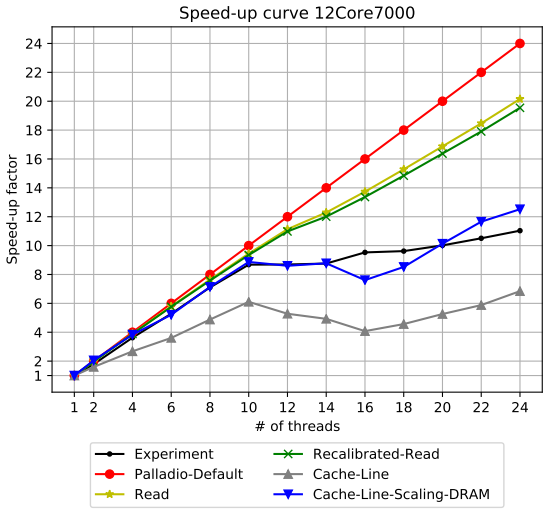
(a) Prediction errors below the black line are under-predictions



(b) Prediction errors below the black line are under-predictions

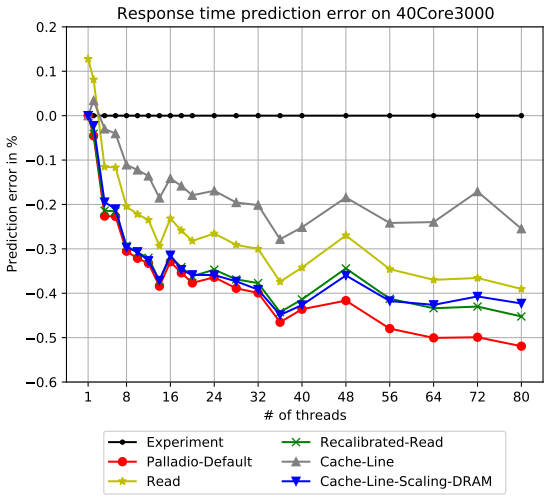


(c) Speed-up curve of measured experiment and the different simulated models

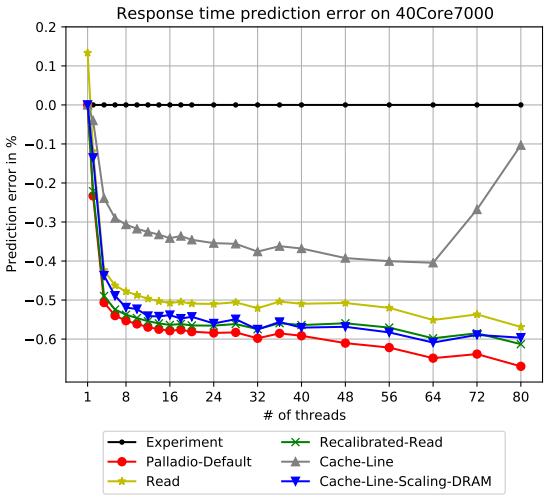


(d) Speed-up curve of measured experiment and the different simulated models

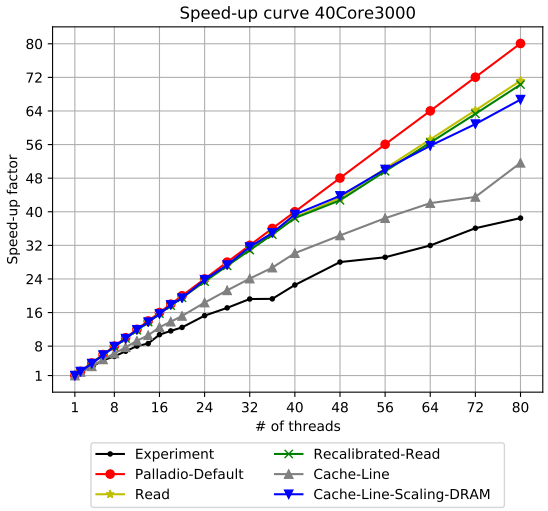
Figure 6.6: Prediction error and speed-up graphs for the 12-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment



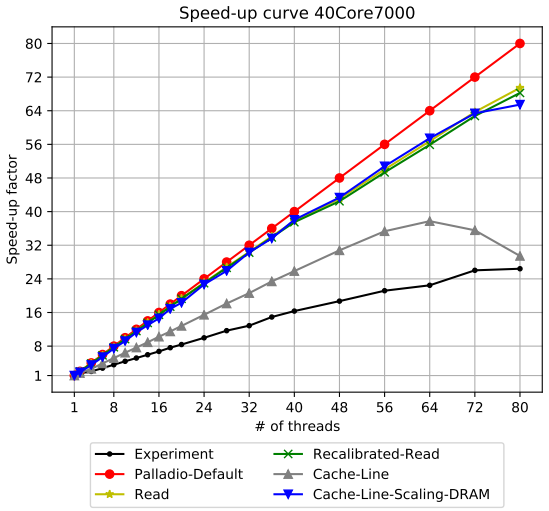
(a) Prediction errors below the black line are under-predictions



(b) Prediction errors below the black line are under-predictions

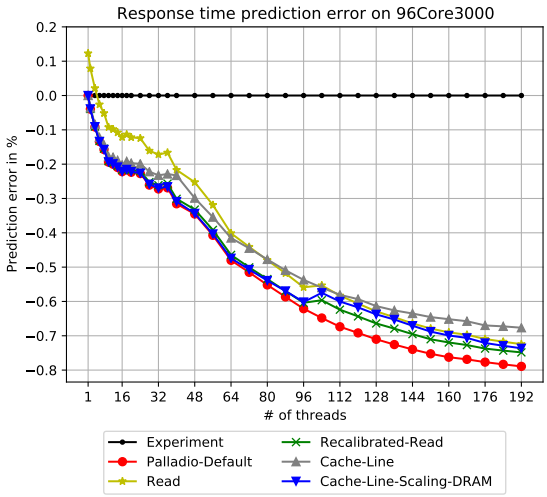


(c) Speed-up curve of measured experiment and the different simulated models

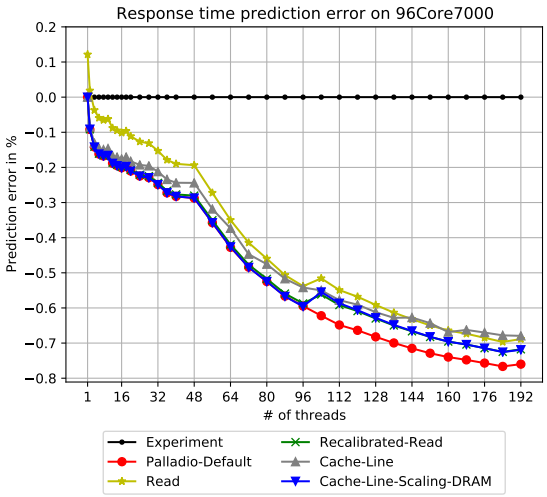


(d) Speed-up curve of measured experiment and the different simulated models

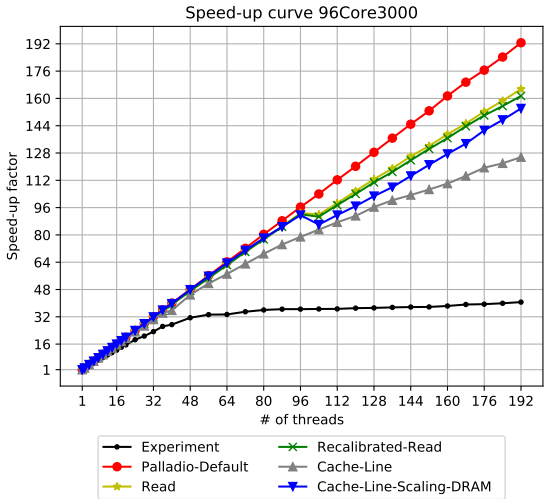
Figure 6.7: Prediction error and speed-up graphs for the 40-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment



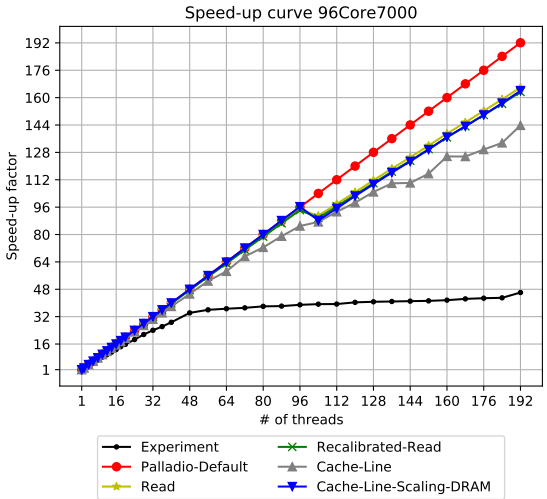
(a) Prediction errors below the black line are under-predictions



(b) Prediction errors below the black line are under-predictions

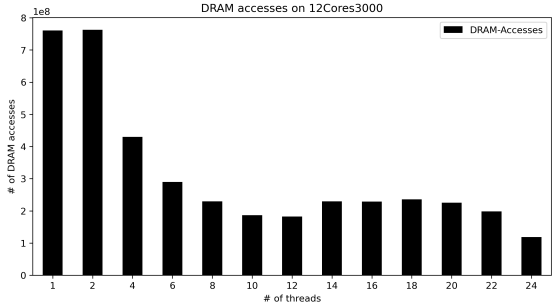


(c) Speed-up curve of measured experiment and the different simulated models

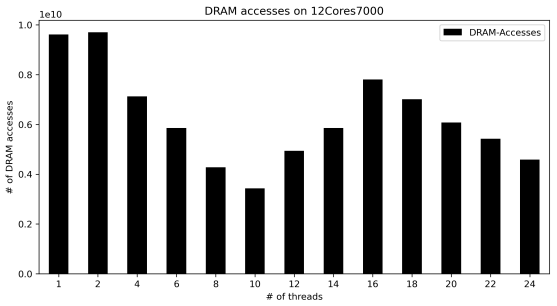


(d) Speed-up curve of measured experiment and the different simulated models

Figure 6.8: Prediction error and speed-up graphs for the 96-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment

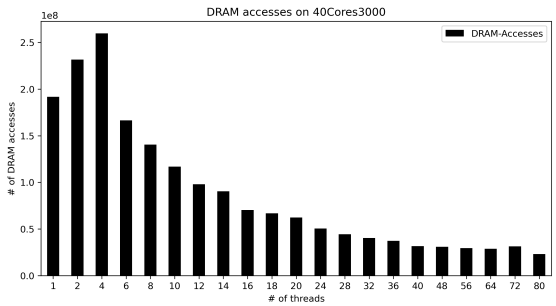


(a) Accesses to DRAM on 12-core server for exp3000 averaged over 100 runs. The used hardware counter is LLC-loads misses

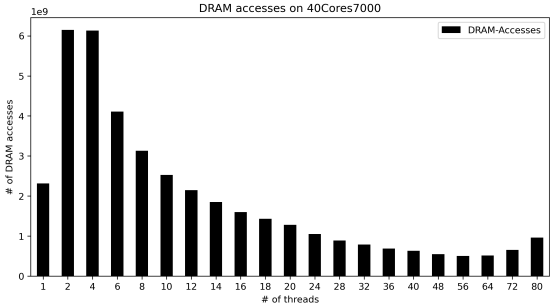


(b) Accesses to DRAM on 12-core server for exp7000 averaged over 50 runs. The used hardware counter is LLC-loads misses

Figure 6.9: DRAM amount of accesses graphs for the 12-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment. Example access values are for 1 thread in exp3000 760 292 434 and for 1 thread in exp7000 9 617 220 819

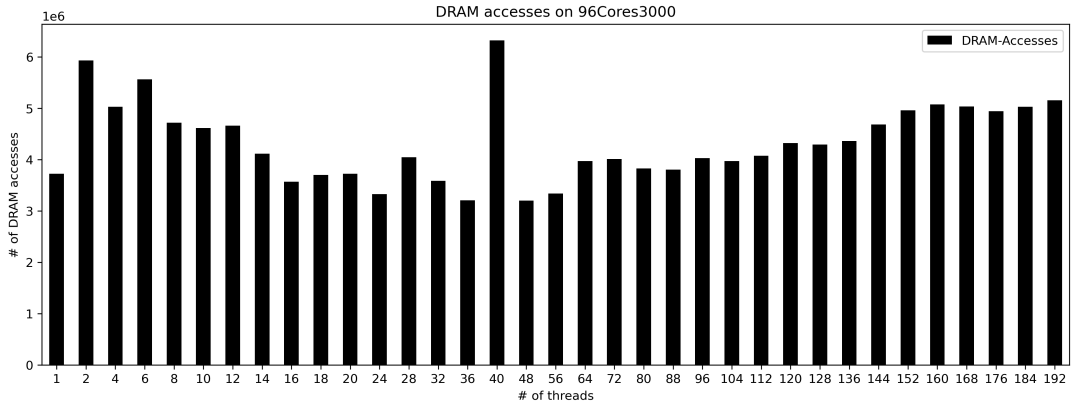


(a) Accesses to DRAM on 40-core server for exp3000 averaged over 100 runs. The used hardware counter is LLC-loads misses

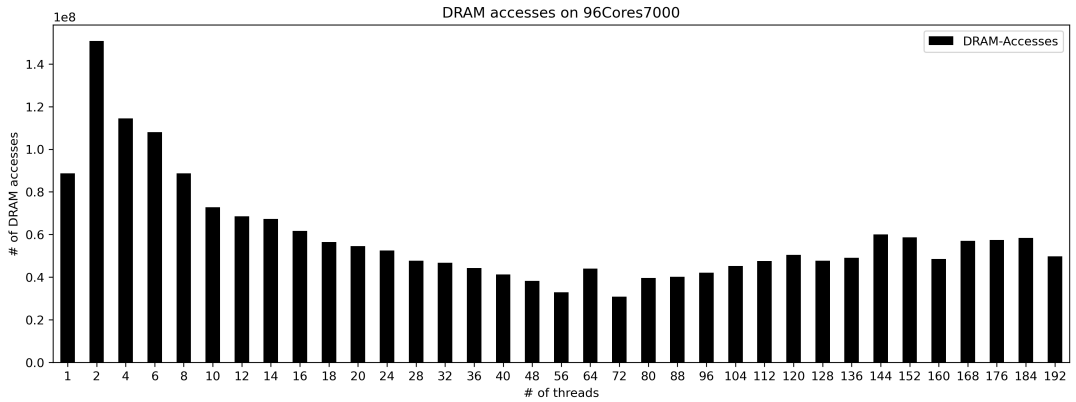


(b) Accesses to DRAM on 40-core server for exp7000 averaged over 50 runs. The used hardware counter is LLC-loads misses

Figure 6.10: DRAM amount of accesses graphs for the 40-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment. Example access values are for 1 thread in exp3000 191 744 586 and for 1 thread in exp7000 2 312 836 256

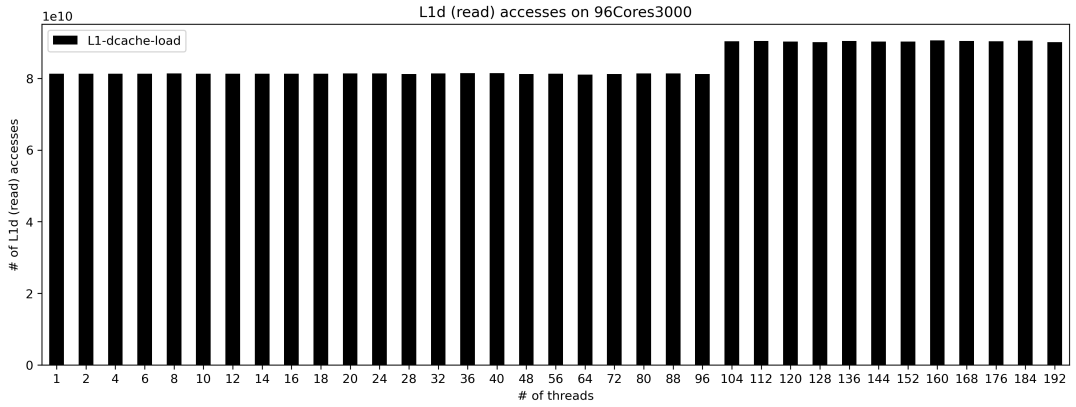


(a) Accesses to DRAM on 96-core server for exp3000 averaged over 100 runs. The used hardware counter is LLC-loads misses

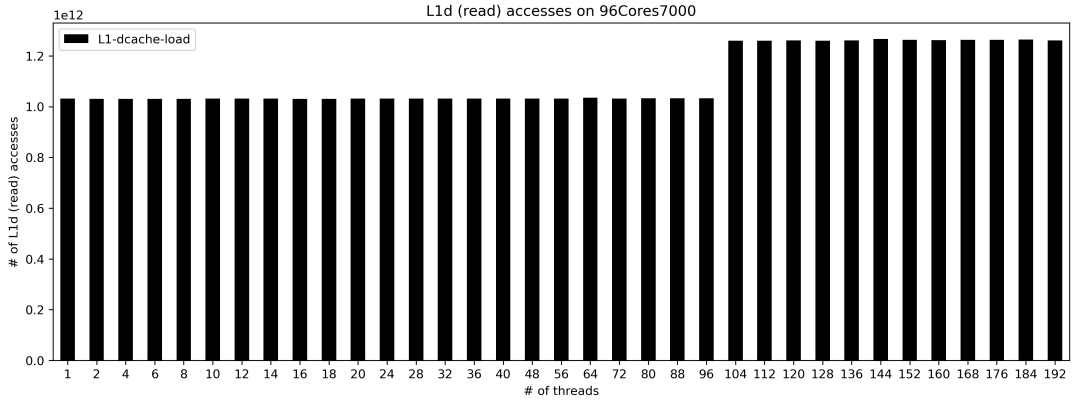


(b) Accesses to DRAM on 96-core server for exp7000 averaged over 50 runs. The used hardware counter is LLC-loads misses

Figure 6.11: DRAM amount of accesses graphs for the 96-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment. Example access values are for 1 thread in exp3000 3 723 672 and for 1 thread in exp7000 88 667 694



(a) Accesses to L1D on 96-core server for exp3000 averaged over 100 runs. The used hardware counter is L1-dcache-load



(b) Accesses to L1D on 96-core server for exp7000 averaged over 50 runs. The used hardware counter is L1-dcache-load

Figure 6.12: L1D read accesses graphs for the 96-core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment. Example access values are for 1 thread in exp3000 81 337 494 779 and for 1 thread in exp7000 1 031 210 265 057

6.6 Threats to Validity

Internal Validity Internal validity describes the validity in our specific experiment setting. For example, there might be effects that influences the independent variables that we are not aware of [WRH+12, p. 106]. The independent variables in the experiment are the measured execution time, cache measurements and bandwidths of the memory hierarchy. The response time prediction is dependent on three factors: the experiment execution measurements of the memory hierarchy, the experiment execution time, and the implementation of the simulation.

The matrix experiment measurements might be a threat to internal validity. The reused matrix experiment might have some fluctuation in its measurements due to some randomness in its execution, e.g., threads were not pinned on fixed cores, therefore, threads could switch to other cores, which might influence the caching behavior. Furthermore, the setup of the matrices did not differentiate between sparse matrices (e.g., matrices filled with a lot of zeros) or dense matrices (e.g., matrices filled with fewer zeros), which could affect the measured execution time in different runs and for the different thread cases. To cope with this randomness, the experiments were repeated multiple times and all measurements were averaged, however, as described in the previous section some irregularities are still observed.

The measurement of cache accesses to calculate cache hit-rates with perf is a further factor that threatens internal validity. In general, performance counter events might differ between different CPU architectures. For example, we also tried to get the write access to the different cache-levels with the *L1-dCache-store* hardware counter event. However,on the 96-core server, the *L1-dCache-store* did not return any result. Furthermore, the read access to L1D with the *L1-dCache-load* event return increasing values form 96 threads and up (see Figure 6.12). However, the measured values of used performance counter events seem to be reasonable as described in Appendix A.3. Additionally, the measurement with perf might alter the way how the matrix multiplication experiments were executed. Therefore, introducing some effects that we are not aware of and influencing the measured execution time.

A further threat to internal validity related to the experiment measurements is Turbo-Boost, which can dynamically change the processor clock rate. The influence of this effect is not investigated.

Simulation implementation flaws while integrating the memory hierarchy into SimuLizar can be a problem. However, by reusing measurements from the previous work of Gruber [Gru19] we got similar results. The small difference to Gruber's approach could be explained (see explanation in 6.4.2) Therefore, the integration should be as good as the modeling and simulation with the existing *LinkingResource*s in Palladio.

External Validity External validity as described by Wohlin et al.[WRH+12, p. 110] describes if the findings can be generalized outside the scope of this study (e.g., industrial practice).

Generalization is limited on the type of processors and its cache design, e.g., all three servers we used for the experiments have Intel processors and a three level cache hierarchy. Therefore, the outcome is unclear for other processors and cache designs. For example, AMD processors and servers with more or less than three cache levels are not evaluated yet.

The matrix multiplication was only measured for a limited amount of thread cases. For example, odd thread cases or thread amounts that are bigger than the amount of virtual cores on each server are not measured.

Furthermore, only a specific algorithm is used to evaluate the influence of memory hierarchy models. It is unclear how accurate the simulation of other algorithms would be. For example, parallel video decoding or sorting algorithms Additionally, an unoptimized version of the matrix multiplication algorithm is used. By using additional code optimization, the implementation can reduce its access to the memory cache hierarchy and also achieve a higher speed-up. Real-Word practice might be different, e.g., parallel code optimization and further optimizations. Moreover, Pyjama, which is an OpenMP-like implementation for Java, is used to parallelize the matrix multiplication. By using other approaches (e.g., self implemented Java threads) the results might differ. In general, the concepts in this thesis might not be ready to be evaluated for real life scenarios.

A further threat to external validity is the experiment measurement setting. The experiment is repeated multiple times and the measurements are averaged. Therefore, we did not investigate the effect of a cold cache, which is the worst case because the caches are empty and all data must be loaded from the main memory. In a real execution, this might occur more frequently. Another limitation to the generalization of our approach is that the measurement is measured in isolation, thus, no other applications or influencing process run in parallel with the experiment. In a real scenario with a real application server, there are maybe other processes that can disturb the measurements for the experiment. In general, contention with other processes is not modeled yet.

7 Conclusion

In this chapter, we first describe the limitations in Section 7.1, then we discuss the research questions in Section 7.2. Afterwards, we list the lessons learned during this thesis in Section 7.3. Finally, we list the future work in Section 7.4.

7.1 Limitations

In the following, we discuss the limitations of this thesis.

Model calibration of CPU The recalibrated model concepts calculated their single thread CPU demand by subtracting the time that was required by the memory hierarchy from the total execution time (see Appendix A.2). It is not clear if it is possible to split the experiment execution time into a CPU only time and a memory hierarchy only time. This relationship between CPU and memory hierarchy is separated to create a simpler model. All recalibrated model concepts that modeled this separation also assume that the speed-up of the CPU part scales perfectly with increasing threads (e.g., for 80 threads the CPU part has a speed-up factor of 80). However, we do not know yet whether there might be some additional influencing factors in the CPU part that causes a slow down with increasing threads.

Additional influencing factors of memory hierarchy missing The following effects that we assume to have an influence on the response time prediction are not included and experimented:

Latency and prefetching: The different model concepts in this thesis do not consider latency effects and did not integrate them yet. It is possible to measure the latency to each cache level and the main memory, however, it is not investigated if data is requested in a perfect stream (e.g., the prefetchers work optimal and the different memory hierarchy bandwidth are completely utilized) or if there is an irregular data access pattern, which leads to a dominating effect of latency. Furthermore, we did not measure prefetching events with perf. Therefore, there might be more data that is transferred through the memory hierarchy.

Links between CPUs: HyperTransport for AMD processors and QuickPath Interconnect for Intel processors are used to communicate between processors. In our models, the amount of transferred data between these links are not measured yet. Furthermore, the bandwidth of these links is not measured yet.

Processor affinity/CPU pinning: How threads are mapped to cores or CPUs might have an impact on the execution time and cache behavior (e.g., if 40 threads are running on 20 physical cores with 40 virtual cores or on 40 physical cores). Furthermore, influences of Hyper-Threading other than the connection limitation are not investigated yet. The missing of CPU pinning (e.g.,

threads should be pinned on specific cores and should not switch during the execution) might also influence the measured values of STREAM multicore scaling bandwidths. The current measurements of DRAM Bandwidth with STREAM did not use CPU pinning. Bandwidth measurement values with CPU pinning might be higher because the accesses are more optimized.

Cache coherency: Caches use a cache coherency protocol such as the MSI protocol or extended versions such as the MESI protocol to keep the data coherent. These protocols add additional traffic to the memory hierarchy that is caused by snooping. These effects are currently missing in the model.

Other neglected influencing factors We assume that further constraints caused by thermal or power limitations might have an impact on the slow down on multicore systems. A further limiting factor is the overhead caused by the parallelization approach with pyjama, which is not investigated yet. Furthermore, the effect of Turbo-boost, which dynamically changes core frequencies, is not evaluated and modeled in this thesis.

Limits of current model The current approach of measuring cache with perf for hit-rates, and with memtest86 and STREAM for the memory hierarchy bandwidths requires the existence of real machines. Further limitations to real world scenarios are already listed in the threats to external validity section (see Section 6.6).

7.2 Discussion of Research Questions

In the following, the aims and research questions proposed in Section 1.1 are answered. The goal of this thesis was to increase the response time prediction accuracy for modeled multicore systems in Palladio. To reach the above-stated goal, the following subgoals were posed:

G1 More accurate performance prediction with Palladio for multicore systems with a focus on response time.

- **G1.1** The PCM meta-model must be extended so that the influence of the memory hierarchy and memory bandwidth on the performance can be modeled.
- **G1.2** The simulation tool (SimuLizar) must be adapted to simulate both influences.
- **G1.3** The Sirius-based Palladio editors must be adapted so that the newly introduced modeling elements can be graphically modeled.

These goals were formulated as research questions and their answers are:

RQ1: To answer *what elements are missing to model memory hierarchy and memory bandwidth*, we refer to Section 4.2.1 in which the initial elements that are required to model the memory hierarchy are listed. However, we could not evaluate the influence of all elements in the scope of this thesis. For example, the writing behavior and latency can be modeled but their calibration and impact on the prediction accuracy is unclear. During the evaluation in Section 6.4, we also identified that the limited linking capacity of Hyper-Threading (e.g., virtual cores

do not have their own links), the transfer of cache lines, and a scaling L2-L3 and L3-DRAM bandwidth seems to affect the prediction accuracy. Therefore, these elements should be modeled additionally.

RQ2: The answer to the question of *how can the PCM meta-model be extended with the identified elements* is based on the reusability of existing elements and on how new elements can be integrated into the existing PCM. Therefore, the following two subquestions answer **RQ2**.

RQ2.1: To answer *can existing PCM elements be reused*, we refer to Section 4.3 in which we evaluated existing PCM elements for their suitability. There were many concepts that can be reused. All elements of the memory hierarchy are derived from existing concepts or elements in PCM. However, most of them must be adapted. The four elements from PCM that are directly reused or adapted are: *ResourceEnvironment*, *ResourceContainer*, *LinkingResource*, and *ResourceCall*. The most differentiating object from PCM is the *MemoryHierarchyLinkingResource*, which have a data flow direction, which is not determined by the component developer, but the system deployer. Furthermore, PCM's *ResourceContainer*s have some similarities with our proposed *MemoryCache*s. The processing concepts of *ResourceContainer* could be reused in the *MemoryCache* in further refinements of the model.

RQ2.2: To answer *what are the trade-offs between the different ways to extend a meta-model*, we refer to Section 4.3.3 in which two alternatives to integrate the memory hierarchy structure are explained. The alternatives are subclassing or profile with stereotypes. The profile with stereotypes approach was chosen because the extensibility support is better. For example, multiple stereotypes can be applied at once. To achieve the same with subclassing is cumbersome. For the call to the memory hierarchy, the answer depends on the extension approach of the simulation as described in Section 4.3.5. Furthermore, the memory hierarchy call extension also has an impact on the Sirius-based editors as described in Section 5.2. Additionally, the way how other simulation tools such as EventSim would treat this call might play a role too.

RQ3 The question of *how can the Palladio tools be adapted to the extended PCM meta-model* is answered by the following subquestions.

RQ3.1: To answer *how to adapt the simulation tool SimuLizar*, we refer to Section 5.1.2 in which four different approaches are evaluated. We recommend either to make changes on the call interpretation level or on the *ScheduledResource* level.

RQ3.2: To answer *how to adapt the Sirius-based Palladio editors*, we refer to Section 5.2. Two adaptions were necessary. The first adaption handles the memory hierarchy structure, which is modeled in a new viewpoint in the Sirius-based Palladio editors. The second adaption handles the memory hierarchy call and is dependent on the simulation extension approach. Either no changes have to be made or a diagram extension of the SEFF viewpoint must be used. However, the diagram extension might have problems with other extensions that also use a diagram extension.

RQ4 To answer *does this extension improve the accuracy of multicore performance prediction compared to previous works*, we evaluated the prediction error of different model concepts in the results Section 6.4. In the results section, the five model concepts Palladio-Default, Read-Data, Recalibrated-Read-Data, Cache-Line, and Cache-Line-Scaling-DRAM were

evaluated. The two model concepts from previous works are Palladio-Default and Read-Data, which are described in Section 6.4.2. Based on the results of Table 6.2, we can observe that nearly all model concepts have a lower mean prediction error than the Palladio-Default model concept. Based on the observations for the Read-Data model concept in Section 6.4.2, there were overpredictions for the first few thread variations. We assumed that the CPU demand calibration might be wrong because the memory single memory hierarchy time is still included in the single thread CPU demand calibration. Therefore, additional simulation delay is mostly caused by this higher initial CPU demand. For exp7000 on the 12-core server in which the L3 to DRAM bandwidth is saturated according to our cache access and bandwidth measurements, the Cache-Line-Scaling-DRAM has a really low mean prediction error compared to all other model concepts.

The final answer to this question is yes, because Palladio-Default does not add a delay at all and the Read-Data model might be wrong calibrated. Furthermore, we assume that the current model concept of Cache-Line-Scaling-DRAM is suited for all cases in which the connection to DRAM is saturated and becomes the bottleneck. However, further elements and model calibration approaches are required to get a more accurate prediction. These are discussed in future work in Section 7.4.

7.3 Lessons Learned

In the following, some lessons learned during this thesis are summarized.

Analyzing prediction for different thread cases each with different measurement require model changes for each thread case. To cope with this massive amount of evaluations, the steps from measuring and calibrating the models should be as automated as possible and repeatable. Therefore, planning ahead to design a pipeline to gather measurements, adding them into the model, and simulating the model is crucial.

Further good practice is to design and execute the experiments as automated as possible. For example, in this thesis, we decided to scale the matrix multiplication experiment from 3000x3000 to 7000x7000 and existing Jar builder scripts made the transition faster and simpler.

The measured cache accesses to DRAM seem to have an impact on speed-up, e.g., as described in the results Section 6.4, most speed-up anomalies of the executed experiments could be explained with the measured access to DRAM. However, these anomalies in the speed-up curve are small and it is still unclear how strong the memory hierarchy influences the speed-up with increasing threads.

The correctness of perf measurements about the memory hierarchy is difficult to judge at first sight. For example, some unexpected measurements on the 96-core server are only noticed after careful evaluation as described in the results interpretation Section 6.5.

A visual representation of the speed-of curve of real measured experiments might be a good visual indicator for the quality of measurements. As described in the results interpretation in Section 6.5, most anomalies in the speed-up curve could be traced back to DRAM measurements.

From the cache access measurements with perf, we saw in the 1 thread case that by scaling the multiplication loop by a factor of approximately 12.7 the L1D and DRAM access also scale by a factor of approximately 12.7. This effect was observed on the 12-core server and 40-core server. Only the 96-core server showed some differences as described in the results interpretation in Section 6.5.

From related work, we know that not only the Core-L1 and L1-L2 connections scale with increasing core usage but also the L2-L3 and L3-DRAM connections. In the proposed modeling concepts, the model concept with Cache-Line-Scaling-DRAM seems not to overpredict in cases in which we assume that the L3-DRAM connection saturates. Without L3-DRAM scaling the overprediction is very high, as it was observed for the Cache-Line model concept.

The LinuxO(1) scheduler has limitations for some CPU demands. For example, we observed that for some CPU demands no delay is added and that for some CPU demands 100ms are added on top of the predicted response time.

7.4 Future Work

The developed memory hierarchy performance model in this thesis has several limitations on its capability. These limitations should be solved in future work.

Finding missing factors by further experiments In the limitation Section 7.1, we listed some influencing factors that we assume to be missing. We assume that the integration of latency and prefetching, and the links between CPUs might increase the accuracy of the predictions. Furthermore, we suggest executing all experiments or measurements with core pinning to gather more stable results. Another factor that might increase the prediction accuracy is to evaluate the influence of power and thermal limitations on the speed-up. Therefore, experiments that provoke these influencing factors should be designed or this experiment might be reused, however, ways to measure these influencing factors must be investigated. A possible example would be to check for algorithms that make optimal use of prefetching, e.g., most data is already prefetched into the caches before it is requested. Therefore, the memory hierarchy connections are used the whole time and latency effects might be reduced. A further influencing factor is the effect of cache coherency protocols with their snooping traffic, however, we assume that the factors mentioned before might have a bigger impact on the speed-up behavior.

Improve modeling constructs The *ResourceCall* that is used to model the transferred data through the memory hierarchy should be integrated into the Parallel Loops template of the Parallel Performance Catalogue [FBKK19], which is already integrated into Palladio. Therefore, manual modeling of massive amounts of threads can be avoided.

Extend the current meta-model of memory hierarchy Some influencing factors were discovered during the evaluation phase, e.g., cache line transfer or the scaling bandwidth of L2-L3 and L3-DRAM connections. These factors are not represented in the current meta-model and should be integrated in a future iteration. The memory transfer demand of the current meta-model takes in

the total amount of memory as bytes, however, this should be changed into the number of read or write accesses instead. Thus, during the simulation all accesses to the lower memory hierarchy can be simulated as cache line size. For the scaling bandwidths, it should be checked how to model connections with a dynamic scaling property, which changes based on the number of thread usages. Furthermore, the isPrivate property of the *MemoryCache*s elements determines if predecessor connections to it scale perfectly or not. This isPrivate property might require changes because of the scaling bandwidths.

Evaluate interaction between schedulers and memory hierarchy The interaction between more specialized CPU schedulers and the memory hierarchy was not investigated in this thesis. We tested the usage of the LinuxO(1) with our models, however, there seem to be limitations that are not investigated yet.

Evaluation of other algorithms In this thesis only the matrix multiplication algorithm is evaluated, however, this algorithm can be implemented in different ways, e.g., more optimized, therefore, improving its cache hit-rates. Moreover, other algorithms with different behaviors should be evaluated.

Evaluation of other computer architectures This thesis only used servers with Intel processors with three cache levels. Other processors (e.g., AMD Processors) with different amounts of cache levels should be investigated in future work.

Increase abstraction level to reduce measurements After sufficient prediction accuracy is reached we would suggest reducing the measurements that are required to calibrate the current model. For example, some analytical approaches that can predict cache hit-rates instead of measuring it can be integrated. Some measurements might be still required, however, these measurements might be simpler to gather.

Bibliography

[BBM13] M. Becker, S. Becker, J. Meyer. “SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems”. In: *Proceedings of Software Engineering 2013*. Vol. P-213. Lecture Notes in Informatics (LNI). Aachen: Gesellschaft für Informatik e.V. (GI), 2013, pp. 71–84. ISBN: 978-3-88579-607-7. URL: http://subs.emis.de/LNI/Proceedings/Proceedings213/71.pdf (cit. on p. 11).

[Bec08] S. Becker. “Coupled model transformations for QoS enabled component-based software design”. PhD thesis. Universität Oldenburg, 2008 (cit. on pp. 10, 11).

[Bec17] M. W. Becker. “Engineering Self-Adaptive Systems with Simulation-Based Performance Prediction”. PhD thesis. Universität Paderborn, 2017 (cit. on p. 11).

[BKR09] S. Becker, H. Koziolek, R. Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22 (cit. on pp. 1, 9).

[BMB+14] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, S. Kounev. “Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures”. In: *IEEE transactions on software engineering* 41.2 (2014), pp. 157–175 (cit. on p. 10).

[BT19] C. Bruns, S. Touati. “Empirical study of Amdahl’s law on multicore processors”. PhD thesis. INRIA Sophia-Antipolis Méditerranée; Université Côte d’Azur, CNRS, I3S, France, 2019 (cit. on p. 19).

[DGI+17] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, L. Sousa. “Modeling large compute nodes with heterogeneous memories with cache-aware roofline model”. In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer. 2017, pp. 91–113 (cit. on p. 18).

[Dre07] U. Drepper. “What every programmer should know about memory”. In: *Red Hat, Inc* 11 (2007), p. 2007 (cit. on p. 12).

[FBKK19] M. Frank, S. Becker, A. Kaplan, A. Koziolek. “Performance-influencing Factors for Parallel and Algorithmic Problems in Multicore Environments: Work-In-Progress Paper”. In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 21–24 (cit. on pp. 1, 4, 87).

[FH16] M. Frank, M. Hilbrich. “Performance Prediction for Multicore Environments—An Experiment Report”. In: *Proceedings of the Symposium on Software Performance*. 2016, pp. 7–9 (cit. on pp. 1, 3, 19, 20, 56, 57, 64).

[FHLB17] M. Frank, M. Hilbrich, S. Lehrig, S. Becker. “Parallelization, modeling, and performance prediction in the multi-/many core area: A systematic literature review”. In: *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*. IEEE. 2017, pp. 48–55 (cit. on p. 17).

[FKB18] M. Frank, F. Klinaku, S. Becker. “Challenges in Multicore Performance Predictions”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. Berlin, Germany: ACM, 2018, pp. 47–48. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3185773 (cit. on p. 1).

[FKHB19] M. Frank, F. Klinaku, M. Hilbrich, S. Becker. “Towards a parallel template catalogue for software performance predictions”. In: *Proceedings of the 13th European Conference on Software Architecture-Volume 2*. 2019, pp. 18–21 (cit. on p. 47).

[FT20] M. Frank, K. T. Truong. *Extending the Palladio Meta-Model to Support Memory Hierarchy*. en. 2020. DOI: 10.5281/ZENODO.4094588 (cit. on pp. 53, 54, 57, 73).

[GJKP13] J. Göbel, P. Joschko, A. Koors, B. Page. “The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development.” In: *ECMS* 13 (2013), pp. 100–109 (cit. on p. 37).

[Gro13] H. Groenda. “Certifying Software Component Performance Specifications”. en. In: (2013). DOI: 10.5445/KSP/1000036063 (cit. on p. 34).

[Gru19] P. Gruber. “Using Palladio network links to model multicore architecture memory hierarchies”. B.S. thesis. 2019 (cit. on pp. 2, 3, 19–23, 25, 27, 28, 53, 56, 57, 64, 65, 81).

[GS13] N. Giacaman, O. Sinnen. “Pyjama: OpenMP-like implementation for Java, with GUI extensions”. In: *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*. 2013, pp. 43–52 (cit. on p. 56).

[Hag] G. Hager. “Performance engineering as a guiding principle for efficient implementations of algorithms in computational science”. In: () (cit. on pp. 12, 18).

[Hap09] J. Happe. *Predicting software performance in symmetric multi-core and multiprocessor environments*. KIT Scientific Publishing, 2009 (cit. on pp. 3–5, 65).

[Hau09] M. Hauck. “Extending Performance-Oriented Resource Modelling in the Palladio Component Model”. In: *Master’s thesis, University of Karlsruhe (TH), Germany (February 2009)* (2009) (cit. on pp. 28, 34).

[HEF15] J. Hofmann, J. Eitzinger, D. Fey. “Execution-cache-memory performance model: Introduction and validation”. In: *arXiv preprint arXiv:1509.03118* (2015) (cit. on p. 18).

[Int20] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Accessed: 2020-10-10. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html (cit. on p. 14).

[IPS13] A. Ilic, F. Pratas, L. Sousa. “Cache-aware roofline model: Upgrading the loft”. In: *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24 (cit. on pp. 17, 18).

[KDH+12] M. E. Kramer, Z. Durdik, M. Hauck, J. Henss, M. Küster, P. Merkle, A. Rentschler. “Extending the Palladio component model using profiles and stereotypes”. In: *Palladio Days* (2012), pp. 7–15 (cit. on p. 32).

[LB05] P. L’Ecuyer, E. Buist. “Simulation in Java with SSJ”. In: *Proceedings of the Winter Simulation Conference, 2005.* IEEE. 2005, 10–pp (cit. on p. 37).

[Leh18] S. M. Lehrig. “Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge”. en. In: (2018). DOI: 10.5445/KSP/1000079766 (cit. on pp. 10, 25).

[McC+95] J. D. McCalpin et al. “Memory bandwidth and machine balance in current high performance computers”. In: *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2.19–25 (1995) (cit. on p. 14).

[Mer08] E. Merks. *Creating Children You Didn’t Know Existed*. Accessed: 2020-10-10. 2008. URL: https://ed-merks.blogspot.com/2008/01/creating-children-you-didnt-know.html?m=1 (cit. on pp. 32, 35).

[Mey11] J. Meyer. “Modellgetriebene Skalierbarkeitsanalyse von selbst-adaptiven komponentenbasierten Softwaresystemen in der Cloud”. In: *University of Paderborn, Masterarbeit* (2011), pp. 1–131 (cit. on p. 11).

[PH16] D. A. Patterson, J. L. Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016 (cit. on p. 13).

[PKG05] B. Page, W. Kreutzer, B. Gehlsen. *The Java simulation handbook: simulating discrete event systems with UML and Java*. Shaker Aachen, 2005 (cit. on p. 11).

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, M. Kuperberg. *The Palladio Component Model*. Tech. rep. Karlsruhe: KIT, Fakultät für Informatik, 2011. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503 (cit. on pp. 1, 33).

[RBH+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, K. Krogmann. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016 (cit. on pp. 1, 4, 7, 9, 10, 26).

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008 (cit. on p. 7).

[SHL+16] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, R. Heinrich. “Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel”. In: (2016) (cit. on p. 29).

[Sta73] H. Stachowiak. *Allgemeine modelltheorie*. Springer, 1973 (cit. on p. 7).

[WFP07] M. Woodside, G. Franks, D. C. Petriu. “The future of software performance engineering”. In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 171–187 (cit. on pp. 1, 8).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on p. 81).

[WWP09] S. Williams, A. Waterman, D. Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76 (cit. on p. 17).

[XCDM10] C. Xu, X. Chen, R. P. Dick, Z. M. Mao. “Cache contention and application performance prediction for multi-core systems”. In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 76–86 (cit. on p. 18).

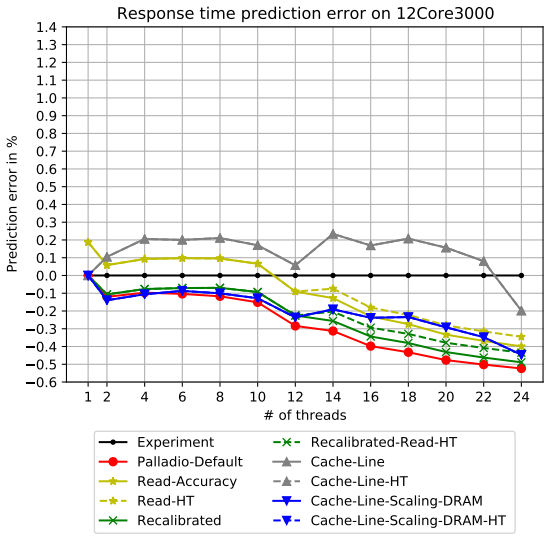
All links were last followed on October 20, 2020.

A Appendix A

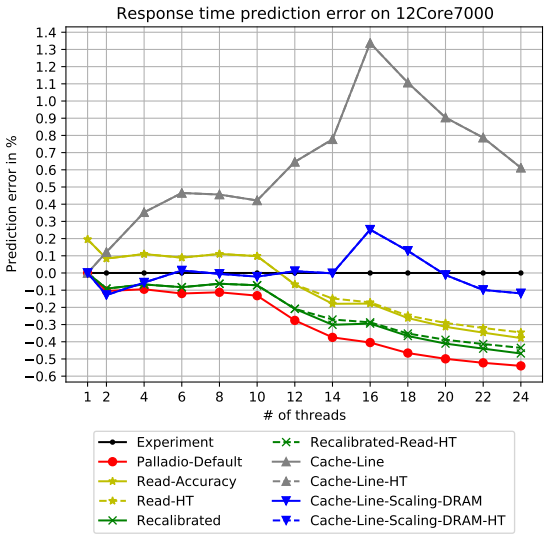
A.1 Additional Data for Hyper-Threading Link Limitations

		Iteration 0		Iteration 1	Iteration 2	Iteration 3
Server	Experiment Variation	Palladio-Default [%]	Read-Data [%]	Recalibrated-Read-Data [%]	Cache-Line [%]	Cache-Line-Scaling-DRAM [%]
12-core	3000x3000	27.1	18.6 (16.2)	23.1 (20.7)	**15.3 (15.3)**	19.6 (19.6)
	7000x7000	28.0	18.6 (17.5)	22.0 (21.0)	61.4 (61.4)	**6.5 (6.5)**
40-core	3000x3000	35.1	27.2 (26.1)	33.5 (32.4)	**15.8 (15.8)**	33.1 (32.3)
	7000x7000	54.3	47.6 (46.9)	52.4 (51.8)	**29.8 (29.8)**	50.6 (50.3)
96-core	3000x3000	43.9	**37.0 (36.2)**	42.6 (41.8)	37.9 (37.9)	42.8 (41.4)
	7000x7000	42.1	**35.1 (34.0)**	41.4 (40.2)	37.5 (37.5)	42.0 (40.4)

Table A.1: Mean prediction error in percent. The smaller the error, the more accurate is the prediction. The values with parentheses are with Hyper-Threading connection limitation modeled, e.g., set their limitation to 12, 40, or 96 based on the server core number. Bold values highlight the most accurate model concept for each row. The mean prediction error is averaged for thread variation described in 6.2. The mean prediction error of the 96-core server can not be used to compare with the 40-core server (see explanation in notes on results in section 6.5).

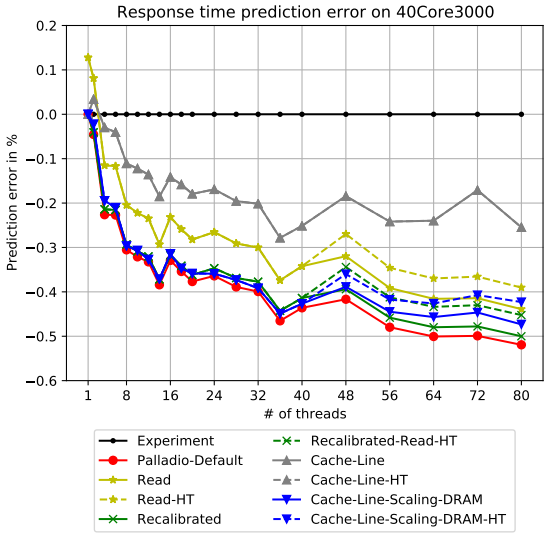


(a) Prediction error of measured experiment and the different simulated models

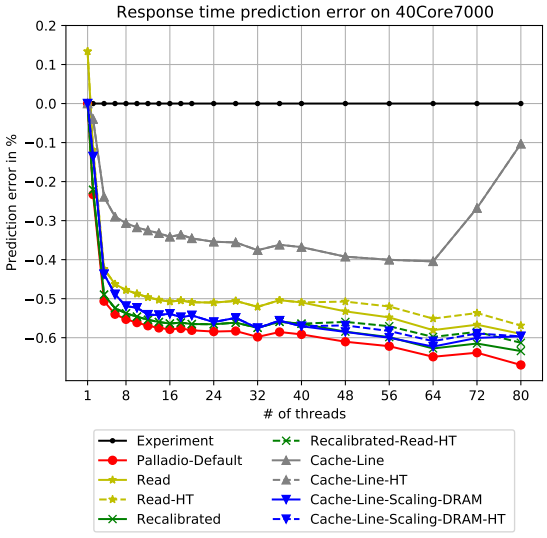


(b) Prediction error of measured experiment and the different simulated models

Figure A.1: Prediction error graphs with Hyper-Threading connection limitation for the 12-Core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment.

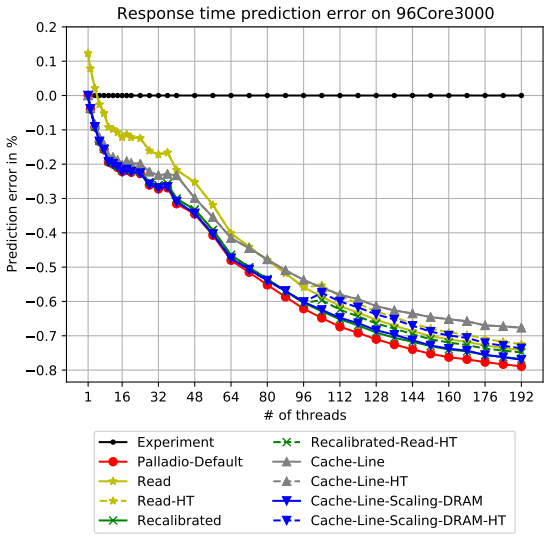


(a) Prediction error of measured experiment and the different simulated models

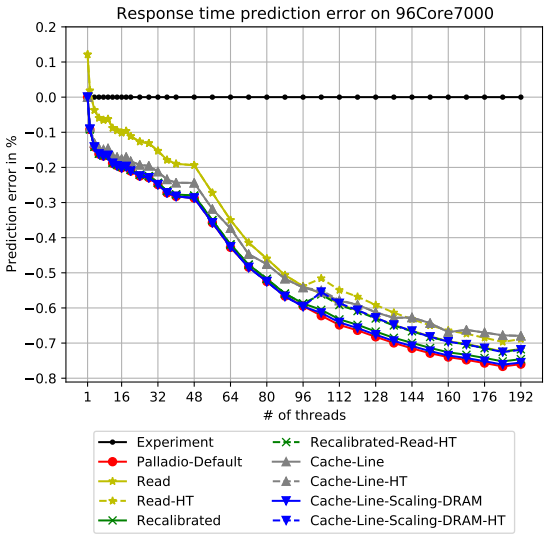


(b) Prediction error of measured experiment and the different simulated models

Figure A.2: Prediction error graphs with Hyper-Threading connection limitation for the 40-Core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment.



(a) Prediction error of measured experiment and the different simulated models



(b) Prediction error of measured experiment and the different simulated models

Figure A.3: Prediction error graphs with Hyper-Threading connection limitation for the 96-Core server. Left are graphs from the experiment in which two 3000x3000 matrices are multiplied with each other. Right graphs are from the 7000x7000 matrix multiplication experiment.

A.2 CPU Calibration Formulas

The recalibrated CPU demand is calculated with the following formula:

(A.1) CPU_only_time = single_thread_execution_time − memory_hierarchy_only_time

The memory_hierarchy_only_time is calculated for model variations that do not use cache line size transfer and instead transfer Java Integers, which are 4 byte, as follows:

$$\text{memory\_hierarchy\_only\_time} = \left(\frac{\text{L1\_dcache\_load} \times 4}{\text{CoreToL1\_Bandwidth}} + \frac{\text{L2\_load} \times 4}{\text{L1ToL2\_Bandwidth}} + \frac{\text{L3\_load} \times 4}{\text{L2ToL3\_Bandwidth}} + \frac{\text{DRAM\_load} \times 4}{\text{L3ToDRAM\_Bandwidth}} \right) \tag{A.2}$$

For a CPU demand recalibration with cache lines the lower cache levels are multiplied with the cache line size instead. In our evaluation, all our test systems have a cache line of 64 bytes. If the cache line size changes, then another value must be used. Therefore, the memory_hierarchy_only_time is calculated as follows:

(A.3)

$$\text{memory\_hierarchy\_only\_time} = \left(\frac{\text{L1\_dcache\_load} \times 4}{\text{CoreToL1\_Bandwidth}} + \frac{\text{L2\_load} \times \text{cache\_line\_size}}{\text{L1ToL2\_Bandwidth}} + \frac{\text{L3\_load} \times \text{cache\_line\_size}}{\text{L2ToL3\_Bandwidth}} + \frac{\text{DRAM\_load} \times \text{cache\_line\_size}}{\text{L3ToDRAM\_Bandwidth}} \right)$$

A.3 Perf Hardware Counters

This section describes the used hardware counters, and the formula to calculate the hit-rate based on access and miss measurements.

On all three servers the following hardware events were used:

For read access :

L1-dcache-loads: This counter describes all read accesses to the L1D cache level.

L1-dcache-load-misses: This counter describes all missed read accesses to the L1D cache level. Furthermore, we used the number of misses as numbers of L2 cache level read accesses.

LLC-loads: This counter describes all read accesses to the LLC cache level, which in our case is the L3 cache level.

LLC-load-misses: This counter describes all missed read accesses to the LLC cache level, which in our case is the L3 cache level. Therefore, this counter describes the number of DRAM accesses.

For write access :

L1-dcache-store: his counter describes all write accesses to the L1D cache level.

L1-dcache-store-misses: This counter describes all missed write accesses to the L1D cache level. Furthermore, we used the number of misses as numbers of L2 cache level write accesses.

LLC-store: This counter describes all write accesses to the LLC cache level, which in our case is the L3 cache level.

LLC-store-misses: This counter describes all missed write accesses to the LLC cache level, which in our case is the L3 cache level. Therefore, this counter describes the number of DRAM write accesses.

All the counters listed above are named events and might differ from processor type or generation.

The *L1-dcache-loads* seems to be reasonable, based on the values. For example, for the experiment with two 3000x3000 matrices, the counter always returned ≈ 81 000 000 000 L1D read accesses, which is reasonable because the matrix multiplication loop executes 3000*3000*3000*3 ≈ 81 000 000 000 read operations. Also, The *L1-dcache-store* seems to be reasonable. For example, for the experiment with two 3000x3000 matrices, the counter always returned ≈ 17 000 000 000 L1D write accesses, which is reasonable because the matrix multiplication loop executes 3000*3000*3000*1 ≈ 17 000 000 000 write operation. We also checked these values for the experiment with two 7000x7000 matrices.

For the *LLC-loads* and *LLC-store*, we also used the *longest_lat_cache.reference* to test their correctness. For example, the *longest_lat_cache.reference* returns all accesses to the last level cache, and *LLC-loads* and *LLC-store* added with each other returned the similar results as the measured

longest_lat_cache.reference value. This is similar for *LLC-loads-misses* and *LLC-store-misses*, and *longest_lat_cache.miss*. However, these counters differ from machine to machine, therefore, we strongly advise to test them before.

The hit-rate is calculated with the following formula:

$$hit\_rate = \frac{total\_accesses - misses}{total\_accesses}$$

For example, if the total L1 accesses is 100 and the misses are 20, then the hit-rate is:

$$\frac{100 - 20}{100} = 0.8$$

A.4 STREAM Scaling Cache Measurements

This section lists the different STREAM measurements for the 12 A.2, 40 A.3, and 96 A.4 core servers. All measurements are from the COPY vector of STREAM.

Threads	STREAM-COPY (in MB)	Threads	STREAM-COPY (in MB)
1	9505000	14	19732000
2	18298000	16	18618000
4	21075000	18	18470000
6	18581000	20	19463000
8	19723000	22	20451000
10	19127000	24	18368000
12	19021000		

Table A.2: Scaling DRAM bandwidth in MB for 12 core server

Threads	STREAM-COPY (in MB)	Threads	STREAM-COPY (in MB)
1	4021000	22	30414000
2	8484000	24	23879000
4	16135000	28	30480000
6	19446000	32	25487000
8	26024000	36	30441000
10	23707000	40	24559000
12	29140000	48	25008000
14	26307000	56	25086000
16	22288000	64	29153000
18	28538000	72	29120000
20	22855000	80	29662000

Table A.3: Scaling DRAM bandwidth in MB for 40 core server

Threads	STREAM-COPY (in MB)	Threads	STREAM-COPY (in MB)
1	12890000	56	184031000
2	25373000	64	184142000
4	48546000	72	172228000
6	73096000	80	182380000
8	90935000	88	185358000
10	115242000	96	187748000
12	111459000	104	169347000
14	131449000	112	172880000
16	150576000	120	173798000
18	169578000	128	190406000
20	155719000	136	192675000
22	155247000	144	203947000
24	169668000	152	216627000
28	183252000	160	228697000
32	182728000	168	237806000
36	179109000	176	248450000
40	200648000	184	250238000
48	189959000	192	275487000

Table A.4: Scaling DRAM bandwidth in MB for 96 core server

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature