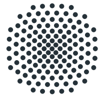


Model-Based Performance Prediction for Concurrent Software on Multicore Architectures—A Simulation-Based Approach

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Markus Kilian Frank

aus Bensheim

Hauptberichter:	Prof. Dr.-Ing.	Steffen Becker
Mitberichter:	Prof. Dr.	Ralf H. Reussner

Tag der mündlichen Prüfung: 15. März 2021

Institut für Software Engineering,
Abteilung Softwarequalität und -architektur
2021

Abstract

Model-based performance prediction is a well-known concept to ensure the quality of software. Thereby, software architects create abstract architectural models and specify software behaviour, hardware characteristics, and the user's interaction. They enrich the models with performance-relevant characteristics and use performance models to solve the models or simulate the software behaviour. Doing so, software architects can predict quality attributes such as the system's response time. Thus, they can detect violations of service-level objectives already early during design time, and alter the software design until it meets the requirements.

Current state-of-the-art tools like Palladio have proven useful for over a decade now, and provide accurate performance prediction not only for sophisticated, but also for distributed cloud systems. They are built upon the assumption of single-core CPU architectures, and consider only the clock rate as a single metric for CPU performance. However, current processor architectures have multiple cores and a more complex design. Therefore, the use of a single-metric model leads to inaccurate performance predictions for parallel applications in multicore systems.

In the course of this thesis, we face the challenges for model-based performance predictions which arise from multicore processors, and present multiple strategies to extend performance prediction models. In detail, we (1) discuss the use of multicore CPU simulators used by CPU vendors; (2) conduct an extensive experiment to understand the effect of performance-influencing factors on the performance of parallel software; (3) research multi-metric models to reflect the characteristics of multicore CPUs better, and finally, (4) investigate the capabilities of software modelling languages to express massively parallel behaviour.

As a contribution of this work, we show that (1) multicore CPU simulators simulate the behaviour of CPUs in detail and accurately. However, when

using architectural models as input, the simulation results are very inaccurate. (2) Due to extensive experiments, we present a set of performance curves that reflect the behaviour of characteristic demand types. We included the performance curves into Palladio and have increased the performance predictions significantly. (3) We present an enhanced multi-metric hardware model, which reflects the memory architecture of modern multicore CPUs. (4) We provide a parallel architectural pattern catalogue, which includes four of the most common parallelisation patterns (i.e., parallel loops, pipes and filter, fork/join, master worker). Through this catalogue, we enable the software architect to model the parallel behaviour of software faster and with fewer errors.

Zusammenfassung

Modellbasierte Performancevorhersagen sind ein bekanntes Konzept zur Sicherung der Qualität von Software. Dabei erstellen Softwarearchitekten abstrakte Architekturmodelle und spezifizieren das Softwareverhalten, die Hardwareeigenschaften und die Interaktion der Nutzer. Sie reichern die Modelle mit leistungsrelevanten Eigenschaften an und verwenden Performancemodelle, um das Software-Verhalten zu simulieren oder durch analytische Methoden zu bestimmen. Auf diese Weise können die Software-Architekten Qualitätsmerkmale wie die Antwortszeit des Systems auf Benutzeranfragen vorhersagen. So können sie Verletzungen der Service-Level-Ziele bereits anhand des Entwurfs erkennen und den Software-Entwurf so lange verändern, bis er den Anforderungen entspricht.

Palladio ist ein Werkzeug, das dem aktuellen Stand der Technik entspricht und sich seit über einem Jahrzehnt bewährt hat. Palladio bietet eine genaue Performancevorhersage nicht nur für anspruchsvolle, sondern auch für verteilte Systeme. Dabei baut Palladio auf der Annahme von Single-Core-CPU-Architekturen auf und berücksichtigt nur die Taktrate als einzige Metrik. Aktuelle Prozessorarchitekturen haben jedoch mehrere Kerne und ein komplexeres Design. Daher führt die Verwendung eines Modells mit nur einer Metrik zu ungenauen Performancevorhersagen für parallele Anwendungen in Mehrkernsystemen.

Im Verlauf dieser Arbeit stellen wir uns den Herausforderungen für modellbasierte Performancevorhersagen, die sich aus Mehrkernprozessoren ergeben, und präsentieren mehrere Strategien zur Erweiterung von Performancevorhersagemodellen. Im Detail diskutieren wir (1) die Verwendung von Mehrkern-CPU-Simulatoren, die von CPU-Herstellern verwendet werden; (2) Wir führen ein umfangreiches Experiment durch, um den Einfluss von leistungsbeeinflussenden Faktoren auf die Performance paralleler Software zu verstehen; (3) Wir erforschen multimetrische Modelle, um die Eigenschaften

von Mehrkern-CPUs besser widerzuspiegeln; (4) und schließlich untersuchen wir die Fähigkeiten von Software-Modellierungssprachen, massiv paralleles Verhalten auszudrücken.

Als Beitrag dieser Arbeit können wir zeigen, dass (1) Multicore-CPU-Simulatoren das Verhalten von CPUs detailliert und genau simulieren. Wenn jedoch Architekturmodelle als Input für die Simulatoren verwendet werden, sind die Simulationsergebnisse von geringer Qualität. (2) Aufgrund der umfangreichen Experimente können wir eine Reihe von Referenzkurven präsentieren, die das Verhalten von charakteristischen Lasten widerspiegeln. Wir haben die Referenzkurven in Palladio integriert und können die Performancevorhersagen erheblich steigern. (3) Wir stellen ein verbessertes multimetrisches Hardware-Modell vor, das die Speicherarchitektur moderner Mehrkern-CPUs widerspiegelt. (4) Wir stellen einen Katalog paralleler Architekturmuster zur Verfügung, der vier der gängigsten Parallelisierungsmuster enthält. Durch diesen Katalog ermöglichen wir es dem Software-Architekten, das Parallelverhalten von Software wesentlich schneller und mit weniger Fehlern zu modellieren.

Acknowledgements

Writing a doctoral thesis is a long and arduous journey with many ups and downs. Mastering the road alone is almost impossible. Therefore, I would like to take a moment and thank all the people who supported me during the past five years.

First of all, I would like to thank my amazing wife, Judith, for her unconditional support, her warm words, and her trust and belief in me. Her constant support did not only eased my mind when I needed to relax, but also gave me the strength to push forward. Deciding to continue my PhD and follow Steffen to Stuttgart forced us back into a long-distance relationship. But this did not stop you from loving me, giving me comfort, and finally marrying me. Judith, I love you more than anything!

No fewer thanks go to my parents: My father Klaus, who is a bright inspiration for me. If it were not for him, I never would have started my PhD and followed his example. Now I hope I am one step closer to being as great as you, Dad. My beloved mother, Anne, was there for me from the beginning. She supports me through all my life and always wants only the best for me. Mum, Dad, you are the best!

Next in line, I have to show my gratitude to all my workmates, first of all, my supervisor Steffen. He moved mountains and fought giants to make it possible for me to start at the university in the first place after my application e-mail was the victim of a spam filter and therefore missed the application deadline. But that is not all: His great feedback always guided me through the foggy journey of my PhD. Thank you for that!

In this context, I would like to thank all my colleagues in Chemnitz. Especially Marcus, who translated and taught me the rules of research as a young researcher. His long discussions helped me to understand the disastrous reviewer comments of rejected papers and helped me to improve. Marcus

is someone to push deadlines with until the last minute, even if that means working all night—or at least as long as there is coffee.

Of course, I also thank all my ”Doktorgeschwister”: Marina, Thomas, Henning, Floriment, Alireza, Vijayshree, Sebastian F., Sandro, Angelika, Stephan, Max, Maximilian, Sebastian K., Heiko as well as the rest of my colleagues André, Jörg, Robert, and Erik for their many hours of discussions, advice, reviews, feedback and research meetings. Especially I would like to highlight here the contributions of Floriment and Alireza—your feedback on the thesis helped me a lot.

Excellent research is not done alone. Thus, a great thanks go to all students I had the pleasure to work with in the past five years. In particular to Denis, Julian, Kim, Lukas, Philipp, and Sebastian. Your high interest in the topics was great, and it was a pleasure to work with you. Finally, I have to admit that writing a doctoral thesis, can be not only hard for the writer, but also his whole environment. In this sense, I would also like to thank all my dear friends for their support, motivational words, feedback, and time they spent drinking beer with me and listening to me grumbling about my work.

Thank you.

Contents

Abstract . . . i

Zusammenfassung . . . iii

Acknowledgements . . . v

I. Introduction and Foundation . . . 1

1. Introduction . . . 3
- 1.1. Requirements for Model-based Performance Predictions . . . 6
- 1.2. Problem Statement . . . 7
- 1.3. Solution Overview & Contributions . . . 7
- 1.4. Thesis Structure . . . 10

2. Foundations . . . 11
- 2.1. Parallel and Concurrent Software . . . 11
  - 2.1.1. Parallel vs. Concurrent . . . 11
  - 2.1.2. Shared vs. Distributed Memory . . . 13
  - 2.1.3. Means to Parallelise . . . 14
  - 2.1.4. Thread-Based vs. Message-Based . . . 18
- 2.2. Single- and Multicore Architectures . . . 19
  - 2.2.1. Architectural Design . . . 19
  - 2.2.2. Common CPU Architecture Example . . . 23
- 2.3. Parallel Programming Patterns . . . 24
  - 2.3.1. Patterns for Parallel Programming . . . 25
  - 2.3.2. Parallel Architectural & Design Patterns . . . 26
  - 2.3.3. Algorithmic Patterns . . . 27
- 2.4. Analyses and Prediction of Quality of Service Attributes . . . 29
  - 2.4.1. CPU Simulators . . . 29

2.4.2. Model-Based Quality-of-Service Predictions on Architectural Level . . . 32
2.5. Hierarchical Queueing Petri Nets . . . 44
2.5.1. Petri Nets . . . 45
2.5.2. Queuing Petri Nets . . . 45
2.6. Summary of Foundations . . . 47

3. Research Design . . . 49
3.1. Research Method . . . 50
3.2. Research Questions . . . 52
3.2.1. RQ_1: Performance Modelling of Parallel Behaviour . . . 52
3.2.2. RQ_2: Behaviour of Highly Parallel Applications . . . 55
3.2.3. RQ_3: Performance Prediction Models . . . 56
3.2.4. RQ_4: CPU Simulators . . . 57
3.3. Research Design Evaluation . . . 58
3.4. Design Science Guidelines . . . 58

4. Related Work . . . 61
4.1. SLR Overview . . . 61
4.2. SLR Planning . . . 62
4.2.1. Research Questions . . . 62
4.2.2. Review Protocol . . . 63
4.2.3. Evaluate Review Protocol . . . 67
4.3. SLR Conducting . . . 68
4.3.1. Executing the Search . . . 68
4.3.2. Applying the Filters . . . 68
4.3.3. Extracting the Data . . . 70
4.3.4. Analysing the Data . . . 70
4.4. SLR Reporting . . . 71
4.4.1. Report Results . . . 71
4.4.2. Evaluate Report . . . 76
4.5. Threats to Validity . . . 78
4.6. Summary . . . 79

5. Running Example: Resource Demands . . . 83
5.1. Resource Demanding Examples . . . 83
5.1.1. Fibonacci Numbers . . . 84
5.1.2. Mandelbrot Set . . . 84

5.1.3. Sorting Arrays 86
5.1.4. Calculating Primes 88
5.1.5. Counting Numbers 89
5.1.6. Matrix Multiplication 90
5.1.7. Summary 91
5.2. Complex Examples 91
5.2.1. Bank Transaction Example 92
5.2.2. SPEC Benchmarks 93

II. Contributions 97

6. Contribution $(CB)_1$: Parallel Architectural Pattern Catalogue 99
6.1. Problem Space 101
6.1.1. General Information 101
6.1.2. Implementation 102
6.1.3. Modelling 103
6.1.4. Experiment Evaluation 104
6.2. Problem Specification - Challenges and Goals 106
6.2.1. Goals 107
6.2.2. Evaluation Metrics 107
6.3. Modelling Language Extension 108
6.3.1. Diagram Types 108
6.3.2. Extension Concepts 109
6.3.3. Diagram and Concept Evaluation 110
6.3.4. Enhancement Process 113
6.4. Proof of Concept Evaluation 118
6.4.1. Result-based Evaluation 119
6.4.2. Goal-based Evaluation 120
6.4.3. Metrics-based Evaluation 121
6.5. Building a Pattern Catalogue 122
6.5.1. Pattern Identification 123
6.5.2. Pattern Categorisation 128
6.5.3. Pattern Selection 128
6.6. Formal Semantics for Parallel Behaviour in the PCM 129
6.6.1. Mapping of general PCM Components 130
6.6.2. Mapping of Parallel Behaviour to Queuing Petri Net (QPN) 133

6.6.3. Evaluation of the Mapping of Parallel Architectural Template (AT)s to QPN 137
6.6.4. Upshot 138
6.7. Empirical Evaluation of the Parallel AT Catalogue 139
6.7.1. Experiment Design 139
6.7.2. Study Conduction 143
6.7.3. Study Results & Reporting 144
6.8. Transferability and Limitations 148
6.8.1. Transferability of the Parallel AT Catalogue 148
6.8.2. Limitations of the Parallel AT Catalogue 148
6.9. Summary of CB_1 149

7. CB_2: Performance Curves for Parallel Behaviour 153
7.1. Problem Space 155
7.1.1. Idea 155
7.1.2. Problem Specification 156
7.1.3. Research Method 158
7.2. Parallel Performance-influencing Factors 158
7.2.1. Parallel Performance-influencing Factors (PPiFs) Collection 158
7.2.2. Prioritising 160
7.3. Experiment-Based Performance Evaluation 161
7.3.1. Experiment Design 162
7.3.2. Experiment Environment 163
7.4. Measurements and Results 165
7.4.1. Result Report Server Stuttgart 165
7.4.2. Comparison of Parallelisation Paradigms 171
7.4.3. Comparison Server 171
7.4.4. Lessons Learned 173
7.5. Extracting Performance Curves 176
7.5.1. Normalisation 176
7.5.2. Clustering 177
7.5.3. Staging 177
7.5.4. Extraction 177
7.5.5. Using Performance Curves: An Example 178
7.6. Palladio Integration 179
7.6.1. Profile Extension 180
7.6.2. Workflow Adaptation 181

7.6.3. OVT-o Transformation 181
7.7. Evaluation 182
7.7.1. Method 182
7.7.2. Results 185
7.8. Assumptions & Threats to Validity 186
7.9. Summary of CB_2 189

8. CB_3: Meta-Model Extension for the PCM to Include Memory Architectures 193
8.1. Problem Space 195
8.2. Meta-Model Extension 196
8.2.1. Meta-Model Elements 197
8.2.2. Meta-Model Extension Strategies 200
8.2.3. Hardware Model Extension 201
8.2.4. Modelling Memory Behaviours 205
8.3. Adaptation of PCM Solvers 207
8.4. Adaptation of Modelling Editors 209
8.5. Evaluation of PCM Extension 210
8.5.1. Experiment Setup 211
8.5.2. Model Calibration 212
8.5.3. Results 215
8.5.4. Result Summary 219
8.5.5. Discussion and Lessons Learned 221
8.6. Threats to Validity & Limitations 223
8.7. Summary of CB_3 225

9. CB_4: CPU Simulators 227
9.1. Problem Space 228
9.1.1. Idea and Goal 229
9.1.2. Problem Specification 229
9.2. Overview of Multicore CPU Simulators 230
9.2.1. Search Strategy 230
9.2.2. Trace-based Simulators 232
9.2.3. Source Code-based Simulators 233
9.2.4. Evaluation and Selection 237
9.3. Palladio Extension Strategies 239
9.4. Trace-driven Strategy 239
9.4.1. SimuCom 241

9.4.2. Discussion 241
9.5. Source Code-Driven Strategy 242
9.5.1. Removal of Java RMI Communication 243
9.5.2. ProtoCom Calibration 244
9.5.3. Discussion 244
9.6. Execution and Use Case Evaluation 244
9.6.1. Use Case and Process 245
9.6.2. Setup 245
9.6.3. Execution & Measurements 246
9.6.4. Discussion 248
9.7. Summary of CB_4 250

III. Evaluation and Summary 253

10. Evaluation 255
10.1. Combination of Contributions 255
10.1.1. Combination of CB_1 255
10.1.2. Combination of CB_2 257
10.1.3. Combination of CB_3 and CB_4 259
10.2. Research Goal Evaluation 260
10.2.1. Answering the Research Questions 261
10.2.2. Assess Requirement Fulfilment 261
10.2.3. Assess the Research Goal Fulfilment 264

11. Conclusion & Future Work 265
11.1. Conclusion 265
11.2. Future Work 267

A. Appendix 271
A.1. Publications & Supervised Theses 271
A.2. Implementations of Resource Demands in Protocom 272
A.2.1. Fibonacci Numbers 272
A.2.2. Mandel Set 272
A.2.3. Sorting Arrays 273
A.2.4. Calculate Prime Demand 274
A.2.5. Counting Numbers Demand 274
A.2.6. Matrix Multiplicationn Demand 275

A.3. User Study Protocols 276
A.3.1. Blank User Study Leaflet—Group A 277
A.3.2. Blank User Study Leaflet—Group B 281
A.3.3. Blank Measurement Protocol 285
A.4. Additional Performance Factor Measurements 286
A.4.1. Speedup Behaviour 286
A.4.2. Cache Behaviour 292
A.4.3. Performance Curves 304
A.4.4. Performance Prediction Error 305
A.5. Memory Hierarchy Models 307
A.5.1. Sirius Extension for Memory Hierarchy Model . . . 307
A.5.2. CPU and Memory Demand Calibration 311
A.5.3. Results HPI Small (12 Cores) 313
A.5.4. Results HPI Large (40 Cores) 314
A.5.5. Results Stuttgart (96 Cores) 315
A.6. CPU Simulator 316
A.6.1. Extension Points to Connect Trace-drive CPU Simulators to Palladio 316
A.6.2. SimulatorBuilder Class 318
A.6.3. MaxSim Config File 318
A.6.4. ProtoCom Calibration 321
A.7. Research Questions and Answers 322

List of Figures 329

List of Tables 333

Literature References 339

Publications of the Author 363

Supervised Theses 369

Introduction & Foundations

1. Introduction

> Manufacturers had been doubling the density of components per integrated circuit at regular intervals, and they would continue to do so as far as the eye could see.
>
> *Gordon E. Moore – 1965*

Software sells. This slogan is true for almost all areas of today's business. Software no longer has a supporting role, but is a core feature and enabler of technology, features, usability, and business. Autonomous cars, smartphones, smart homes, legal tech companies, and multimedia streaming services are only a few examples of successful applications that dominate our daily life and highly depend on sophisticated software. This software is so complex that it contains thousands or even millions of lines of code, it cannot be developed by a single person anymore, and it has to fulfil high levels of quality standards to meet the Service-level Objective (SLO). Due to the complexity of the software and the immense cost of software failures and bugs, such software is developed in an engineering-like way, to ensure high quality standards [KBAW94; KKB+98]. This engineering-like way includes a structured method of collecting requirements, creating architectural designs, as well as evaluating and testing. In the following, we focus on the evaluation of architectural designs used in the early design phase. Therefore, model-based performance prediction approaches are used to simulate and to evaluate the quality attributes of architectural design (e.g., response time). To use such approaches, the Software Architect (SA) must create an architectural model of the software (i.e., the software model), specify the users' behaviour (i.e., user model), and create a description of hardware characteristics (i.e., the hardware model). In the next step, the SA uses simulation-based or analytic

solvers to evaluate the different quality attributes of the architectural design. State-of-the-art approaches like the Palladio Bench[1] or CloudSim[2] achieved accurate predictions for complex, distributed, and cloud systems.

Nevertheless, all of the current approaches consider only a single metric in the hardware model—the CPU speed—as relevant for estimating the performance of the system. This assumption is appropriate when using hardware powered by CPUs with up to four cores. However, today's common CPUs have more than four cores. By now, multicore processors have been widely used for more than a decade in all types of devices, such as smartphones, laptops, and desktop PCs. While smartphones have up to 8 cores, desktop PCs with 16 cores or servers with more than 100 cores are a common sight today.

Moving from single-core CPUs to multicore CPUs brings a range of new challenges to the software engineering domain. First of all, to use the full potential of multicore processors, software developers must write software that supports parallelism on multiple levels. Writing parallel software is even more challenging when the developers must consider live-/ deadlock, synchronisation, concurrent data access, etc.

Different domains tackle the multicore challenge in their ways: In safety-critical embedded systems like aeroplanes or cars it is important to prove the correctness of the application and to guarantee deadline (e.g., detecting and reacting before crashing into an obstacle). Because parallelism significantly increases the complexity, it was common sense in the embedded domain to disable all but one core and continue to use sequential applications [KSS+17]. However, due to the increased amount of software (and thus, hardware requirements) manufacturers are now forced to develop new approaches to not only develop parallel applications but also to specify and verify them.

In the HPC domain, parallel execution has been researched for years. Thereby HPC focuses on low and algorithmic levels. It is common sense to use programming languages like Fortran and to optimise each instruction. So, developers in HPC search for potential optimisations and count each byte to, e.g., fit their instructions into a single cache page. That way, they can gain

[1] http://www.palladio-simulator.com/

[2] http://www.cloudbus.org/cloudsim/

massive performance boosts, since each instruction is executed millions of times.

The developers of Business Information System (BIS) usually do not have the expert knowledge of HPC developers, nor do they have the resource limitations embedded systems have. So, a common practice today is to slice applications into (micro-)services and try to avoid parallelism within these services. Moreover, parallelism is achieved by running multiple instances of a service and handling user requests in isolation. E.g., to coordinate or exchange data in a Kubernetes Cluster, key-value databases like etcd.io are used, even though a shared in-memory solution like Redis might be much faster. But also more complex to handle parallel accesses.

Slicing applications along the user requests (jobs) has the advantage that each job can be handled independently. However, the benefits are limited. With multiple jobs accessing the main memory at the same time—even if they have an isolated memory space—shared resources like the memory bus become a bottleneck. Further, slicing is often not possible due to the domain. E.g., for data analysis, the whole data set is evaluated, and the algorithms are complex, time and resource-intensive. Thus, to use the full potential of today's hardware, the code within the services also needs to be parallelised.

Today, it is still common practice for people from High Performance Computing (HPC) and BIS to follow a try-and-error approach to see if the software under development fulfils the SLOs. This approach is not only cost, and time-intensive, but simply not applicable for large-scale systems like Facebook, Netflix, or Twitter any more. These systems are so large and have such a high number of user requests that it is simply not possible to generate the load for testing any more [WS03].

Thus, having reliable software performance predictions of parallel applications in multicore environments in more critical than ever. Thereby we need to enable SAs to factor in parallel behaviour during the early design phase, which is challenging, since commonly-used languages for designing software (e.g., UML 2.5[3]) have only limited capability to express parallel behaviour and the SA needs to model each behaviour manually. Next, we

[3]UML 2.5 Specification: `https://www.omg.org/spec/UML/2.5.1/PDF`

must reevaluate the model-based performance prediction methods to show their accuracy and suitability for parallel software.

In the course of this thesis, we will not focus on challenges when coding parallel applications but focus on the modelling and performance prediction aspect.

1.1. Requirements to Enable Model-based Performance Predictions for Parallel Software on Multicore Environments

Given this background, we can identify major requirements to successfully perform model-based performance predictions for parallel software run in multicore environments:

$R_{modelling}$ Software architects must be able to express concurrency in software models, that (a) describe the behaviour of the software and (b) is feasible for the SA to model regarding time and effort. This includes highly concurrent software, which can consist of multiple hundreds or even thousands of concurrently executed threads, where it is not feasible to model each thread by hand.

$R_{metrics}$ Since the single metric—CPU speed—is no longer sufficient to cover all the performance-relevant aspects of multicore systems, the software architect must be able to specify the additional performance-influencing factors (e.g., memory bandwidth, cache behaviour, or the memory architecture) needed.

$R_{performance}$ The performance prediction models must include relevant performance-influencing factors and reflect the additional complexity.

$R_{solvers}$ The solvers, used to interpret and analyse the models, need to be capable of processing and evaluating the adapted software, hardware, and performance models.

$R_{accuracy}$ The performance predictions need to align with the real and measurable behaviour of the software to an extent that is useful for the software architect.

1.2. Problem Statement

Unfortunately, no approach exists which fulfils all of the above requirements [FHLB17]. However, approaches exist that meet at least one requirement, although none of them focuses on model-based performance predictions. In what follows, we give a brief overview (see Chapter 4 for a full discussion on the related work):

Memory Architecture Modelling: For $R_{metrics}$ and $R_{performance}$, there are few research projects that use memory architecture modelling to predict the behaviour of the memory [THW09; THW12; VE11; Wil09; XCDM10]. These approaches focus on CPU caches and their hit rates.

Parallel Behaviour Modelling: For $R_{modelling}$, [RGD11b] aims at using UML MARTE profiles to enrich software models with multicore information. However, they do not focus on performance predictions, but on code generation for OpenCL.

Reusable Architectural Knowledge: The Architectural Template Method aims to provide reusable architectural templates to SA [Leh18; LHB18], which can help us address $R_{modelling}$ and $R_{metrics}$.

Due to the paucity of all-encompassing approaches, SAs are currently limited in their ability to model parallel behaviour, and the process of modelling is highly error-prone and time-consuming [FH16; FSH17]. Furthermore, when it comes to performance predictions, SAs are currently not able to make reliable Quality of Service (QoS) predictions for parallel applications running in multicore environments. Thus, an engineer-like approach to develop highly parallel applications suffers from single-metric hardware models, incomplete performance models, inaccurate solvers, and the absence of language support for modelling parallel software behaviour at the moment.

1.3. Solution Overview & Contributions

To overcome the shortages named above, we propose an approach containing four individual contributions combined into the Palladio Bench.

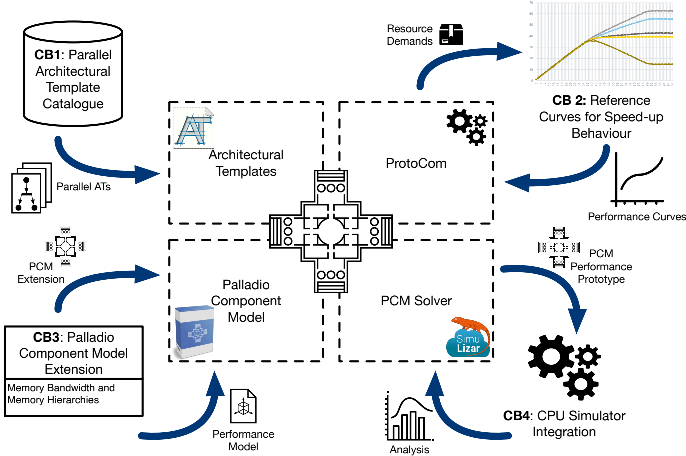


Figure 1.1.: Overview of the solution and contributions presented in this thesis

Figure 1.1 gives an overview of the contributions. To be able to provide them to the SA as a combined approach, we integrate them into the Palladio Bench. In this way the SA can benefit from the full potential of all contributions.

CB_1: First, we provide a parallel architectural template catalogue based on the AT method to offer SAs a set of easy-to-use common parallel design patterns ($R_{modelling}$). Thus, we can significantly reduce the time a SA needs to model parallel behaviour—while keeping the number of errors low. At the same time, we increase user acceptance and improve the user experience. In total, we support four abstract parallel design patterns (Master-Worker, Parallel Loops, Fork & Join, Pipes & Filters), which the SA can use to model the behaviour of 33 common parallel design patterns.

CB_2: We conduct extensive experiments to analyse the impact of performance-influencing factors on the response time ($R_{metrics}$). We use the measurements to derive performance curves, which we integrate into Palladio to increase the prediction accuracy ($R_{accuracy}$). As result, we provide a set of performance curves for common types of software behaviour to the SA. These performance curves can increase the performance predictions without detailed modelling of all performance relevant aspects.

CB_3: We extend the Domain Specific Language (DSL) of the Palladio Bench, the Palladio Component Model (PCM)[BKR09], and include characteristic elements to reflect the memory hierarchy into performance models ($R_{metrics}$, $R_{performance}$). In doing so, we also extend the current state-of-the-art simulator (SimuLizar) to handle the models ($R_{solvers}$). As a result, we present a memory hierarchy model, implement the approach in the PCM and SimuLizar, and are now able to simulate cache behaviour and memory bandwidth utilisation to a certain extent.

CB_4: We connect multicore CPU simulators used by hardware architects and CPU vendors to Palladio. We use the PCM models as input for the simulators, simulate them, and play the results back ($R_{accuracy}$, $R_{solvers}$). We eventually provide two strategies: A trace-driven and a source code-driven approach. We evaluate both methods and are able to show that CPU simulators cannot be used for realistic model-based performance predictions, due to the low-level information needed as input model. This information is absent in our architectural input models.

In the context of this doctoral project, we published a number of peer-reviewed publications including conference papers, journals, workshops, and posters. Further, a number of student theses were supervised by the author of this thesis. Appendix A.1 gives a detailed overview of the publications and topics.

1.4. Thesis Structure

The remainder of this thesis is structured in three parts: Introduction & Foundations, Contributions, and Summary. Table 1.1 gives an outline of the remaining chapters.

Part	Chapter	Content
Part I. – Foundations	2: Foundations	Describes the fundamental concepts needed to follow the thesis. This includes knowledge of CPU architecture, parallel applications and execution, and model-based performance prediction.
	3: Research Design	Here we explain in-depth the research objectives, questions, and method we follow in the thesis.
	4: Related Work	In this chapter we perform a Systematic Literature Review (SLR) to reveal existing approaches to build upon and discover related work.
	5: Running Examples	In the course of the thesis, we use reoccuring code examples, which we will introduce here. We utilise these use cases to provide sample scenarios or evaluate our results later.
Part II. – Contributions	6: CB_1 Parallel Architectural Pattern Catalogue	In order to overcome the lack of parallel language concepts, we create a parallel architectural pattern catalogue in this chapter. As input, we use 35 patterns we discovered in a structured literature review. After categorising, we use the Architectural Template Method to create a catalogue including the four most common parallel patterns. Finally, we evaluate the catalogue using a use-case example an empirical user study.
	7: CB_2 Performance Curves	We conduct extensive experiments to evaluate the influence of specific factors on the speedup behaviour of applications. We extract performance curves from the measurements and include them in the parallel AT catalogue.
	8: CB_3 Memory Model	Here, we discuss memory architectures and their mapping in performance models. We present an extension to the PCM language to include caches and memory bandwidth in the models and extend the SimuLizar simulator to improve the prediction accuracy.
	9: CB_4 CPU Simulators	To further increase the prediction accuracy, we investigate the use of hardware multicore CPU simulators. We research how we can use the PCM instances as input for the simulators and evaluate the overall performance.
Part III. Summary	10: Goal Evaluation	We evaluate the achievement of the research goal, discussing each research question and its answer.
	11: Conclusion	Concludes the thesis by summarising the findings from the contributions, discussing the lessons learned, and suggesting future research.

Table 1.1.: Overview of the thesis structure

2. Foundations

In the following section, we introduce the fundamental concepts needed to understand and follow the rest of the thesis.

First of all, we are going to lay out the basics of parallel and concurrent software. In the same section, we will introduce two different taxonomies to categorise concurrent and parallel software: categorisation based on memory usage and categorisation based on information exchange.

After we understand the software characteristics of concurrent and parallel software, we will expound the hardware characteristics of multicore CPUs. Thereby, we will focus on high-level concepts needed to follow the rest of the thesis.

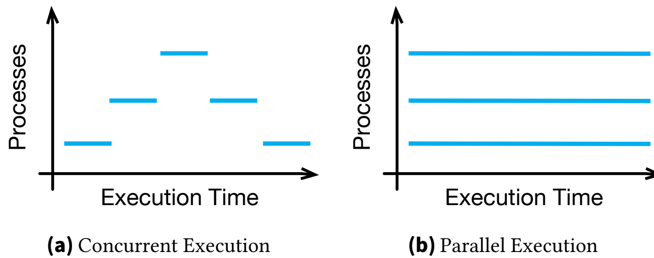
In the latter portion of this section, we will use that knowledge to elaborate common parallelisation patterns, approaches to predict the behaviour of multicore CPUs, and model-based approaches to predict quality attributes of software designs.

2.1. Parallel and Concurrent Software

In this section, we will elaborate on the characteristics of parallel and concurrent software. Thereby, we will focus only on the software view (the hardware view is illuminated in Section 2.2).

2.1.1. Parallel vs. Concurrent

Parallelism and concurrency are often used as synonyms in the literature. However, they are not the same thing.



(a) Concurrent Execution **(b)** Parallel Execution

Figure 2.1.: Concurrent vs. Parallel Execution

In computing, concurrency was first used to better utilise or share resources in a computer (comp. [MSM04]). For that, the computing task is partitioned into smaller subsets and, with the help of the operating system's schedulers, tasks can quickly be swapped. This has the benefit of one task not having to lock the processor while idling (i.e., while waiting for I/O). By quickly swapping many tasks, it appears to the user as if the tasks are executed in parallel. However, this must not be the case. Figure 2.1a exemplifies a concurrent execution of multiple tasks.

Compared to concurrency, parallelism describes the behaviour of two tasks being executed at the same time, in parallel. Figure 2.1b exemplifies a parallel execution.

Finally, we can conclude with the following definitions for concurrency and parallelism from [Sun08]:

"**Concurrency:** A condition that exists when at least two threads are making progress. A more generalised form of parallelism that can include time-slicing as a form of virtual parallelism.

Parallelism: A condition that arises when at least two threads are executing simultaneously."

In addition, Table 2.1 summarises the different characteristics.

While we use concurrency to utilise a single core more efficently, parallelism needs real multicore systems to execute different threads in parallel. Thus, we use multicore systems to improve the throughput of a system. In this thesis, we focus on parallelism, parallel software, and multicore architectures.

	CONCURRENCY	PARALLELISM
Act	It is the act of managing and running multiple computations at the same time.	It is the act of managing and running multiple computations simultaneously.
Method	Interleaving operation	Using multiple CPUs
Benefits	Increased amount of work accomplished at a time.	Improved throughput, computational speed-up
Uses	Context switching	Multiple CPUs for operating multiple processes.
Required Processing Units	Single or multiple	Multiple

Table 2.1.: Comparison of Concurrency and Parallelism (cf. [Tec17])

2.1.2. Shared vs. Distributed Memory

In parallel systems, it can be necessary to exchange data among the individual tasks. Most common approaches are based on either shared or distributed memory approaches. In this context, the terms shared and distributed memory do not refer to the physical location or layout of the memory, but rather to how the memory is presented to the parallel applications (cf. also shared and distributed memory computer architectures):

Shared Memory: In shared-memory approaches, each task can access the whole memory of the application. The data exchange occurs by multiple threads accessing the same data in the memory. This approach is vulnerable to a high number of drawbacks. So, in every parallelisation approach which is based on shared memory (i.e., threads) the developer has to take care of synchronisation, mutual exclusion, and data privacy aspects (cf. [MMG+09]).

Distributed Memory: In comparison to shared memory, distributed memory grants each task access to a specific address space only. Hence, it is not possible for a task to directly access the data of another task. To enable data exchange among tasks, one task has to send data to another task individually to exchange data.

2.1.3. Means to Parallelise

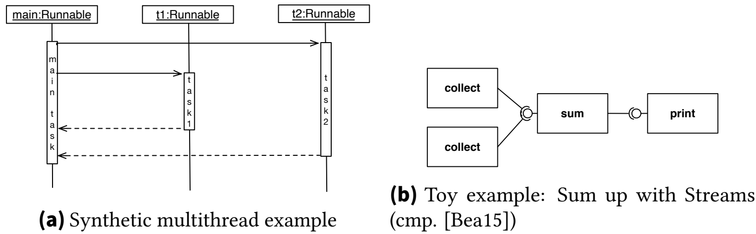
Depending on the given memory access method (shared or distributed), different parallelisation paradigms have to be used to support the access method or to ensure it. In the following section, we will explain two general means of achieving parallelisation: Thread-based and message-based. For each of these two methods, we will give examples of commonly used implementations. The list of examples is far from complete and is only used to explain the basic concept.

2.1.3.1. Thread-Based Approaches

In thread-based approaches, parallelisation is achieved by spawing new threads. The operating system then schedules the new threads to the processors and cores. Data exchange is done by the principle of shared memory, which makes it also necessary for the developer to take care of mutex. In the next three paragraphs, we explain pure threads and stream processing, as well as OpenMP.

Threads: Threads are the most basic means of achieving parallelisation. Figure 2.2a exemplifies the approach. To achive thread-based parallelism within an application, the main thread of the application forks new threads and assigns tasks to them. Each thread executes its subroutine, and by scheduling the threads to individual cores (by the operating system), the threads run in parallel. This approach is often also called task parallelism because each task is separated into an individual thread [Rei07]. To successfully use this approach, it is essential that the individual tasks have no limited, well designed inter-thread communication. If they share the same data, the developer needs to take care of data access restrictions (i.e., locks and mutual exclusion). Thread-based means to parallelise are the foundations for design pattern (like master-worker pattern) or parallelisation patterns (like fork-join). We discuss these patterns in more detail in Section 2.3.

Stream Processing or data-flow programming (sometimes also referred to by the architectural style: pipes and filters) is a programming paradigm well known from Linux command line shells (pipe) or graphical calculations



(a) Synthetic multithread example

(b) Toy example: Sum up with Streams (cmp. [Bea15])

Figure 2.2.: Abstract overview of threads and stream processing

within GPUs. The basic concept is explained by using Figure 2.2b. In stream processing, there is a sequence of data (stream) and a series of operations. The operations are applied in a specific order to the streams to get the desired result set [Bea15]. Each operation is thereby independent of the others, and only needs specific input data. Thus, it is possible to run each operation in parallel and even to have multiple instances of each operation. Typically, each operation instance runs in its thread to archive the parallel execution.

While stream processing traditionally used kernel functions, such as operations, and was optimised for particular CPUs (e.g., GPUs), the concept is widely adopted nowadays, used in common programming languages (e.g., Java Streams), and runs on general-purpose CPUs [GR04].

OpenMP: The OpenMP Application Programming Interface (API) is a precompiler, who was designed by a group of software and hardware manufacturers. Both interest groups have agreed on specifications to create a uniform standard for programming parallel computers with a shared address space. The three main components of OpenMP are compiler directives, runtime libraries, and environment variables. Implementations are available for almost all common programming languages, which makes OpenMP a popular API for developers.

The parallel programming model of OpenMP is based on parallel threads which have a shared and a private address space. All programs start with a single master thread. Based on the fork-join execution model, it creates a so-called thread team. The compiler directive triggers the creation of the team

at the beginning of the program section, which is about to be parallelised. All threads of the team execute this section in parallel. The exchange of information between the threads takes place using shared variables. These variables are kept in the address space shared by all threads concerned. However, private variables are stored on each thread's stack and are therefore only held for the duration of the execution of the parallel section. The shared and private variables are specified in the compiler directives. When parallel processing is completed, the created threads terminate, leaving only the master thread.

OpenMP provides various mechanisms for coordinating the threads. It is possible to implement critical areas in which only one thread may process. To synchronise the threads of a team, OpenMP uses the barrier directive to wait for all threads, and to synchronise the workflow. The barrier directive causes all threads reaching it to pause until all threads of the team have reached it. The programming model also provides locking mechanisms in the form of simple and nestable lock variables. Their use and further implemented concepts for thread coordination are described in detail in [RR12](p.369-373).

One of the most critical aspects of the underlying programming model is the possibility of establishing parallelism on the loop level. Within a parallel section, loops can be parallelised using the for-directive. For this purpose, the loop iterations, and thus the computing work, is distributed to the threads of the team. This distribution can be done in different ways, e.g., by assigning a certain number of iterations to the threads in the team. Another variant is to assign the iteration blocks dynamically. Whenever the processing of a block is completed, a new one is assigned. To use OpenMP parallel loops, the parallel loop must fulfil certain conditions. One is that the total number of iterations must be known before entering the loop. Furthermore, the individual calculations of the iterations must be independent of each other and must not change the running index of the loop (cf. [HL08; RR07]).

Due to its relatively simple use, OpenMP is frequently used to speed up and parallelise legacy software, by merely annotating for-loops, so that they run in parallel.

2.1.3.2. Message-Based Approaches

Message-based parallelism is characterised by the clear intercommunication of a set of concurrent tasks. These tasks may reside on the same physical device, or across an arbitrary number of devices. To exchange data with each other, the tasks communicate by sending and receiving messages. This data exchange usually requires the cooperation of each process [GHK+13]. Even though message-based parallelism approaches can be used on the same machine, message-passing is often associated with distributed memory models and distributed computing.

In the following, we will briefly explain two common frameworks for message-based parallelisation: MPI and Actors.

MPI: Message Passing Interface (MPI) is a specification for developing parallel programs that communicate with each other by the exchange of messages [BVS13]. It is a standard interface for message-passing calls and is powerful, flexible, and usable [SAB18]. One property of MPI is that it is very explicit, meaning that the programmer can control many details of the data flow [Eij17]. Additionally, interface specifications have been defined and implemented for C/C++ and Fortran [BVS13]. Nowadays, MPI has become a standard for developing message-passing applications [BVS13].

Actors: The Actors Model (Actors) is an abstract model for parallel processing. It was first presented in the paper [HBS73], which introduced the basic concept of actors. There are numerous programming languages and partly identically named implementations, which use the axioms of the actors model to implement parallelism, but differ in detail. In the following, we will, therefore, only deal with the core axioms of the actors model:

Actors are considered to be basic, abstract units, which include processing, memory and communication. Actors follow the principles of object-oriented design. Accordingly, actors can be considered as objects, and are encapsulated from each other. The encapsulation also means that no two actors share the same memory. Thus the exchange of information between the actors must take place via explicit communication. Explicit communication happens by the asynchronous exchange of messages (in many implementations

also a synchronous option is available). The actor can react to a message only with three actions:

- Creating additional actuators
- Sending messages to known actuators
- Adjust the behaviour for processing the next message

Actors have a message queue in which the incoming messages are held (see Figure 2.3), since they can only be processed one after the other. Messages are taken from the queue and processed according to the "First In - First Out" principle. Also, the concept of state machines is supported. The state of the actor after processing a message determines the behaviour for processing the next one [Ver15]. Due to encapsulation and independence, actors can be executed in parallel. However, the actors themselves operate like a sequential application. A manual implementation of locks and mutexes is not necessary, because each actor has its own memory space [Cli81].

When it comes to determining potential actors in an application, Storti gave the following statement: "Everything is an actor" [Sto15]. In practice, however, this leads to too much complexity and performance losses for a fine-granular actor system. Therefore, one tends to represent each functional task by an actor [Ver15].

2.1.4. Thread-Based vs. Message-Based

The shared memory model characterises thread-based parallelism. Each thread has its local memory, but also shares the global set of variables. The communication between the threads is achieved by updates and access to memory in the same address space [GHK+13]. Thread-based approaches can be faster than message-based approaches because of the more convenient access to the shared memory address space. However, this shared access can lead to problems, such as race conditions. Message-based approaches have better scalability than thread-based approaches because of the distributed memory model, which enables the simple addition of new parallel tasks. Also, since each task has its isolated memory, race conditions are a much

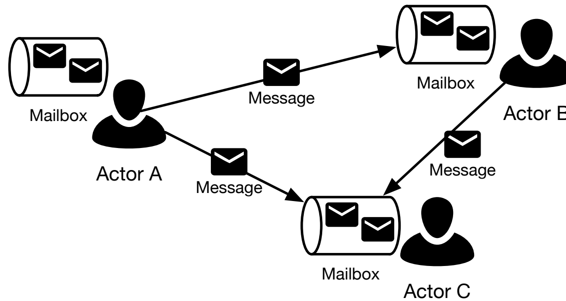


Figure 2.3.: Example of an Actor System (cf. [Doy14])

smaller threat. A disadvantage of message passing is the necessity of implementing an interface that is responsible for the data transfer between the tasks [Pie16].

2.2. Single- and Multicore Architectures

In practice, a wide variety of multicore CPU architectures exist. The variance ranges from very specialised architectures (like GPUs or embedded control units), over networks clusters, symmetric multiprocessors, and massive parallel supercomputer CPUs to off-the-shelf CPUs [MSM04].

In the following, we will give an overview of the most common CPU architectures. A basic understanding of the hardware will later help to understand performance characteristics and performance issues of parallel applications.

2.2.1. Architectural Design

While there are multiple taxonomies to categorise CPU architectures, by far the most common one is the taxonomy introduced by Flynn [Fly72], which we will follow in this section. Flynn categorises all CPU architectures by the

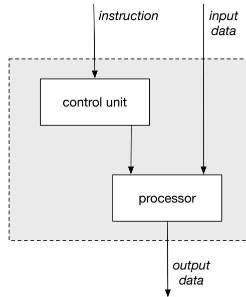


Figure 2.4.: Example of SISD (cf. [MSM04])

number of instruction streams and data streams. Thereby, a stream is a sequence of instructions or data a CPU processes. Flynn distinguishes between four different types: SISD, SIMD, MISD and MIMD [Fly72; MSM04].

Single Instruction and Single Data (SISD): Figure 2.4 exemplifies the category of SISD. In this type, each processing unit in the system gets its own data for each instruction.

Single Instruction and Multiple Data (SIMD): In contrast to SISD, in SIMD systems, each processing unit gets the same instruction but different data upon which to execute the instruction. A typical example is image processing or digital signal processing, which are well suited for low-level parallelism (see Figure 2.5).

Multiple Instruction and Single Data (MISD): For MISD there is no well-known system that fits this category, and it is only included in Flynn's taxonomy for the sake of completeness.

Multiple Instruction and Multible Data (MIMD): In a MIMD system, each processor unit has its own set of instructions and its own set of data upon which to execute the instructions (see Figure 2.6). Each processor unit has an interconnection bus to exchange information with the other processors. This group of systems is the most generalised one while at the same time, fitting modern multicore architectures the best.

Only considering Flynn's taxonomy is a good start. However, it is not sufficient for understanding multicore architectures as a whole. In particular,

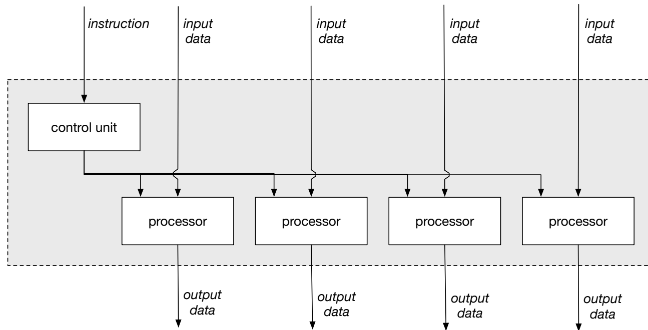


Figure 2.5.: Example of SIMD (cf. [MSM04])

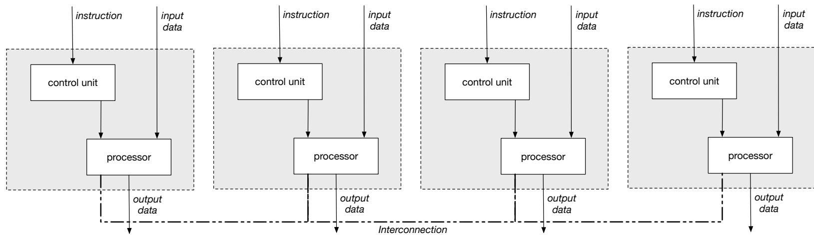


Figure 2.6.: Example of MIMD (cf. [MSM04])

the memory hierarchies and the CPU core interactions are not detailed enough. Thus, Mattson et al. [MSM04] specified additional subcategories for MIMD: Symmetric Multiprocessors (SMP) and Non-Uniform Memory Access (NUMA) architectures.

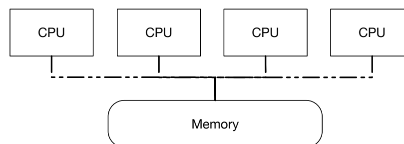


Figure 2.7.: Exemplification of SMP (cf. [MSM04])

SMP: Figure 2.7 shows the composition of SMP. It is a subclass for shared memory systems. Each CPU accesses the same memory, while only

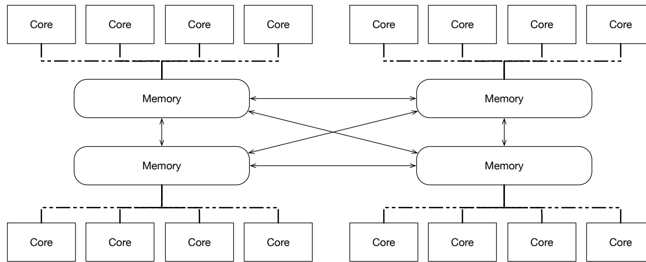


Figure 2.8.: Exemplification of NUMA (cf. [MSM04])

one memory exists in the architecture. Furthermore , all CPUs share the same connection (memory bus) and can access the memory at the same speed. SMP architectures are the easiest for the programmer, because there is no need to consider the location of the data. In this kind of architecture, the memory bus often becomes a bottleneck, because the utilisation of the bus increases with an increasing number of cores. Therefore, this architecture does not scale well, and only works for a limited number of CPUs.

NUMA: A more complex architecture is NUMA architecture, which Figure 2.8 illustrates. As in SMP architectures, the memory is shared, and each processor can access all blocks in the memory. However, some blocks of memory might be more closely associated with some CPU cores than others. Thus, cores can access data located in a closer memory faster and therefore, the access times for data located in different memories can differ significantly. To compensate for these effects, a hierarchical cache system is often used [KTJ06] together with a strategy to maintain cache-coherence. Hence, these architectures are also called cache-coherent nonuniform memory access systems (ccNUMA).

For the sake of completeness, we also have to mention the subcategories for distributed-memory architectures. In a distributed-memory architecture, each processing unit has its memory and address space (see Figure 2.9). Communication with the other processors is done by message passing. Depending on the topology, the communication speed can range from as fast as shared memory to rather slow (e.g., communicating over an ethernet

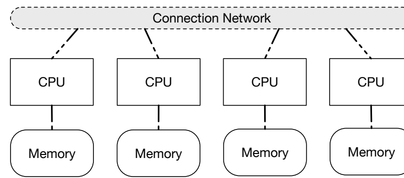


Figure 2.9.: Exemplification of a Distributed Memory Architecture (cf. [MSM04])

network) [MSM04]. Even though these kinds of systems have a high research interest, especially in the domain of HPC, we will focus in this thesis on general-purpose CPUs since the business information applications we are interested in use this kind of hardware architecture.

2.2.2. Common CPU Architecture Example

To foster understanding, we will briefly describe the architecture of a common general-purpose CPU with a hierarchical memory hierarchy (like an Intel i7) in this section using Figure 2.10. In Figure 2.10, multiple processors are depicted. Each processor contains multiple cores. Common desktop processors currently have 2 to 32 cores per processor (i.e., AMD Ryzen Threadripper 3970X[1]).

Each core contains a Central Processing Unit (CPU) and two types of Level 1 Cache (L1)—one for instructions (L1 Instruction cache) and one for data (L1 data cache). The L1 cache is directly accessible by the CPU and guarantees fast access of data in case of a cache hit. Further, each core has its Level 2 Cache (L2), which is, in comparison to the L1 cache, slightly larger, but its access times are slower.

Depending on the system's architecture and the mainboard used, multiple processors can be used. Thereby, the memory bus connects the individual processors with the Last Level Cache (L3) and the main memory. If there is too much communication between processors, or between processors and main memory, the bus can become a bottleneck—similar to network links.

[1] https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x

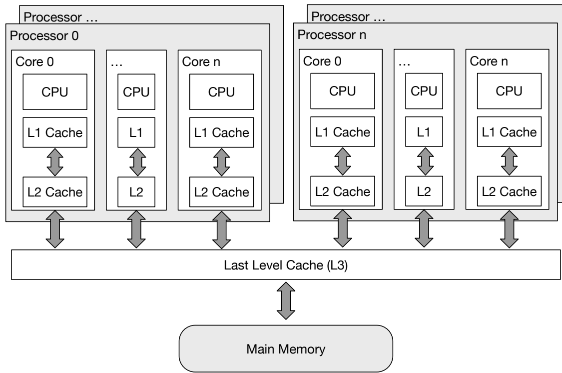


Figure 2.10.: Example of a Common Hierarchical Multicore Processor (cf. [Sch08])

Often mainboards support prioritised access from one processor to a specific segment of the main memory (RAM module), which improves the access rates of data in that segment.

Besides architectures with hierarchical memory hierarchies, there are also architectures with a pipeline or array design [RR07]. However, since they are not common, we will skip explaining them at this point.

2.3. Parallel Programming Patterns

In the past years, not only specialised domains like HPC, but also standard application developers and researchers have had to face the need for efficient parallel software. However, developing such software is complex, challenging, and error-prone [MMG+09]. Therefore, a quite broad range of best practices and patterns has arisen to guide developers when realising parallel software.

In this section, we will introduce fundamental parallelisation patterns. Understanding the pattern will help to comprehend the core concepts of parallel programming. Following this section will also help to elucidate contribution 1 (see Chapter 6), in which we introduce a parallel pattern catalogue for common modelling languages, such as UML2.

First, we look at the pattern definition. Afterwards, we will introduce different categories of parallel patterns and explain the main concepts behind them.

2.3.1. Patterns for Parallel Programming

Mattson et al. defines a pattern as follows:

> "A (design) pattern describes a good solution to a recurring problem in a particular context. The pattern follows a prescribed format that includes the pattern name, a description of the context, the forces (goals and constraints), and the solution. The idea is to record the experience of experts in a way that can be used by others facing a similar problem. In addition to the solution itself, the name of the pattern is important. It can form the basis for a domain-specific vocabulary that can significantly enhance communication between designers in the same area." [MSM04, p. 11]

Starting from this, defining the characteristics of a pattern is tricky, fuzzy, and in practice, the gap between a pattern description and its implementation can significantly differ. Further, the same pattern often goes by different names in different communities. Therefore, we performed a literature review in [SWD19] to categorise common parallel patterns and find synonyms. The main results of this review are shown in figure 2.12, and a more detailed discussion is given in Section 6.5.1.

After collecting parallel patterns from the literature, we extracted the description and grouped similar patterns together, naming the pattern by the most common name (i.e., fork & join). Further, we categorised the patterns by their level of abstraction, into three groups: Algorithmic, Architectural, and Design Patterns (Figure 2.12 groups the latter two, for reasons of simplification). For each pattern, Figure 2.12 lists synonyms or implementation variants. This list is far from complete and is intended only to provide a rough overview.

In the following, we describe each main pattern in more detail.

2.3.2. Parallel Architectural & Design Patterns

"[An architectural] pattern provides a scheme for refining elements of a software system or the relationships between them. It describes a commonly recurring structure of interacting roles that solves a general design problem within a particular context." [BHS07, p. 392]

2.3.2.1. Master-Worker Pattern

According to [Eij17], the master-worker pattern is one of the most well-known patterns in parallel programming, and is supported by a broad set of programming languages. The basic idea behind the master-worker pattern is simple: One mammoth task is split into multiple subtasks that can run in parallel. Thereby it is essential that the subtask is as isolated as possible, in order to avoid interdependences.

The master is in charge of distributing the work to the workers, as well as coordinating them.

Since it is a design pattern, the master and the workers are often designed as individual components. While each worker-component has a specific task, the master-component takes over the role of a facade and a load-balancer or task manager. The calling instances call only a function on the master-component, which also provides the result to the calling instances.

On a lower abstraction level, this pattern behaves similarly to the fork/join pattern.

2.3.2.2. Message Passing

We already discussed in Section 2.1.3.2 the basic idea of message passing: each acting instance has its own memory, and can only interact with other instances by sending messages. Often these messages are sent asynchronously, and each acting instance has a message queue to store messages until they can be processed [Erb12].

To implement the message passing pattern, languages that support these features are required. One option is to use object-oriented programming

languages to manually implement the pattern, frameworks like AKKA Actors (for Scala)[2], or specific languages like Erlang[3].

Choosing a message-passing approach is a fundamental design and therefore categorised here as an architectural pattern.

2.3.3. Algorithmic Patterns

Algorithmic patterns are, in contrast to design and architecture patterns, on a much lower abstraction level and focus on a solution strategy for one concrete implementation problem. An algorithmic pattern, therefore, describes a solution strategy with one or multiple subroutines.

In the following three paragraphs, we will describe three parallel algorithmic patterns. All of them are based on shared memory, and have a thread-based approach.

2.3.3.1. Parallel Loops & Sections

Parallel loops are an efficient way to realise parallelism for programs that show a need for many repetitions of the same calculation without dependency between loop cycles [MSM04].

Its ease of achieving parallelism defines the parallel loops pattern, and it requires a set of independent data that can be split into smaller subsets. Each data subset is initially passed to an individual loop. E.g., considering a list of 800 entries, where for each entry the same operation is performed. The parallel loop pattern would split the list into, e.g., four subsets—each subset containing 200 entries. Now instead of having one single loop iterating over 800 elements, we have four loops iterating over 200 elements each. By separating each loop into an individual thread, parallelism can be achieved. To achieve the best results, splitting the dataset into equal and independent parts is critical.

[2] https://doc.akka.io/docs/akka/current/typed/index.html
[3] https://www.erlang.org/

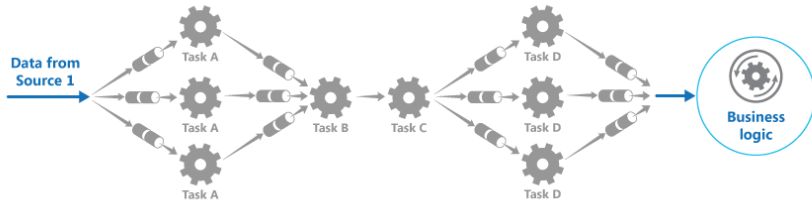


Figure 2.11.: Example Stream Application (cf. [Mic17])

2.3.3.2. Streams or Pipes and Filters

The pipes & filters pattern is rather common as well. The following description is a summary of the official Microsoft Azure documentation [Mic17].

The pipes & filters pattern can be used as a parallelism approach based on data streams. A stream consists of filters, which are processing steps, and pipes that represent connections between filters.

The pipes & filters pattern works by separating a set of data into streams and applying pipelines of pipes and filters in a predetermined order onto these streams. While each filter is independent of the others, and only relies on the input stream, parallelisation can be achieved by executing different filters in parallel. Slow filters can have multiple instances to faster process the input stream. In the end, the processed data stream is collected. Figure 2.11 illustrates this approach.

2.3.3.3. Fork-Join

The following content is based on information found in [Eij17] and is very similar to the master-worker pattern. Even though the abstraction level is much lower, the idea is the same: due to the logical identification of subtasks, one mammoth task is split into subtasks, which can be executed in parallel.

In the best case, the subtasks are independent. However, this is often not the case. Therefore, locks and synchronisation mechanisms are used to include barriers, mutually exclusive data access, and waiting conditions.

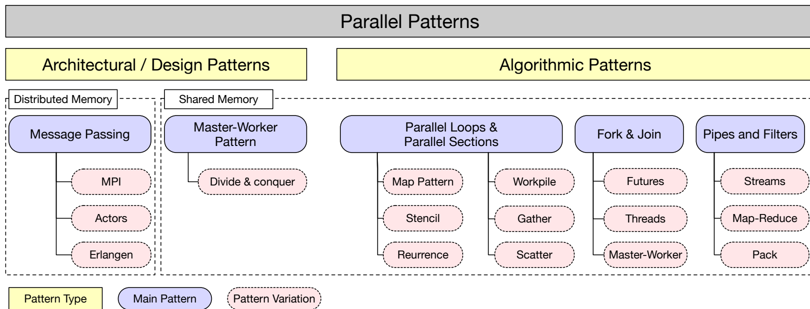


Figure 2.12.: Categorisation of Parallel Patterns

2.4. Analyses and Prediction of Quality of Service Attributes

The analyses and prediction of QoS attributes (e.g., response time) is a major part of Software Performance Engineering (SPE). C. Smith and W. Lloyd define SPE as following: "*SPE is a model-based approach that uses deliberately simple models of software processing with the goal of using the simplest possible model that identifies problems with the system architecture, design, or implementation plans. These models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals.*"[SW03]

In this section, we will first introduce an approach using CPU simulators to estimate the QoS attributes. Second, we focus on the model-based QoS predictions on architectural level (e.g., the Palladio approach).

2.4.1. CPU Simulators

CPU simulators are often used by hardware vendors to evaluate the quality attributes of new CPU architectures. However, they can also fulfil various other duties. The primary duty we are interested in is the estimation of quality attributes of a parallel software running on a target environment without deploying it. So, one of the biggest challenges is the consideration of

different types of CPU architectures. Even though common architectures are supported by now, and CPU simulators deliver reliable results, the simulation takes much time.

In the following, we will describe the main characteristics of CPU simulators. CPU simulators are relevant to follow the accomplishment in Contribution 4 described in Section 9. Parts of this section originated from the collaboration with a Student—S. Graef [Gra18].

2.4.1.1. Foundations of CPU Simulators

Hardware architects have researched CPU simulators for years. The main difference is in the type of entry, the calculation method and the application scope [AS16].

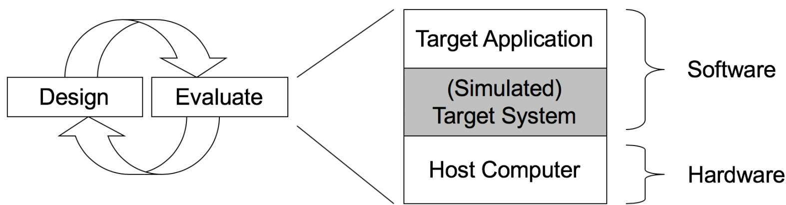


Figure 2.13.: Simulation of target components [Carl J. Mauer – Computer Sciences Department – University of Wisconsin]

Figure 2.13 shows the evaluation process when using simulators. The interesting part here is the target application, which is running on the target system. While the target application is known, the target system needs to be simulated (or emulated) on the host machine (i.e., the computer running the simulation) [AS16].

In the following sections, we describe the different dimensions according to [SK13a] in detail:

Section 2.4.1.2: Functional vs. Timing Simulators

Section 2.4.1.3: Cycle-Driven vs Event-Driven Simulators

Section 2.4.1.4: Trace-Driven vs. Execution-Driven Simulators

Section 2.4.1.5: User-Level vs Full-System Simulators

In Chapter 9, we perform a literature survey and use the above dimensions to classify the CPU simulators. Thus each simulator receives a trade-off (spider-web) diagram, which briefly describes its characteristics.

2.4.1.2. Functional vs. Timing Simulators

The group of functional simulators are used for functionality testing [AS16] only. Thus, they are not relevant in this theses. Because we do not research the correctness of application, but the behaviour and performance.

In contrast to functional simulators, timing simulators focus on the exact behaviour. They can simulate the hardware and software under study to an extent, that it is possible to get performance counter for any time. Most timing simulators are also called cycle-level simulators [AS16], because they track every clock cycle. The cycle level accuracy, however, comes on the cost of time. The simulations times of cycle-level simulators are up to 25 times longer than functional simulators, and they use more compute resources [AS16].

2.4.1.3. Cycle-Driven vs. Event-Driven Simulators

To further drill down into the group of timing simulators. We can distinguish between two additional subgroups: the cycle-driven (cycle-accurate, or cycle-level) and the event-driven simulators.

While cycle-driven approaches are relatively slow, event-driven simulators reduce the time consumption. One particular kind of event-driven simulators is interval simulators [GEE10]. Interval simulators combine the feature set of functional and timing simulators. But they do not simulate on cycle-level but in intervals. The idea is that the missing events such as branching, mismatches, and cache misses dividing the normal command flow through the pipeline into intervals. Then these intervals are evaluated separately. This combination can reduce simulation time [AS16].

2.4.1.4. Trace-Driven vs. Execution-Driven Simulators

CPU simulators use a different kind of inputs. We distinguish between trace-driven and execution-driven. Trace-driven simulators use a trace as input. The traces contain detailed and low-level information about the execution. One drawback is that trace files can grow very large. But on the plus side, it is not necessary to emulate the Instruction Set Architecture (ISA) with this type of simulator [AS16].

In contrast to that, execution-driven simulators use an executable application as input. When it comes to accuracy execution-driven simulators are very accurately by emulating the ISA and also take errors that occur into account (e.g., incorrectly specified code path) [AS16]. Thus, this type of simulators is most suited to predict the behaviour of an application.

2.4.1.5. User-Level vs Full-System Simulators

User-level (or application-level) simulators do not consider operating system calls. In contrast, full-system simulators take the system calls into account. So, the predictive power for system calls intensive applications is better with full-system simulators. The disadvantage is that the simulators become heavy [AS16].

2.4.2. Model-Based Quality-of-Service Predictions on Architectural Level

In model-driven software development, models are used to develop the software on a high abstraction level, which abstracts the software‘s complexity to ease understanding and analysability. As a result, models become a central artefact and are used for, e.g., code generation and automatic deployment.

In an early design phase, models are used to analyse and improve the software before the software is realised. SPE is such an analysis method. SPE aims to predict the software's quality attributes, such as response time (performance), costs of operation (costs), and range of capable performance (scalability) [BDIS04]. Later, this approach was used in model-driven performance engineering, which allows software developers to design performance models in

a DSL [BDIS04; Hap08]. However, to derive performance metrics from such models, it is necessary to combine model-based hardware descriptions with software descriptions and environment/usage descriptions.

The main advantages of SPE are that software developers are able to evaluate the performance requirements of the system at an early stage. In this phase, decisions and design can easily be altered, because no realisation has to be adapted. So different design alternatives can be evaluated and compared, and trade-off decisions can be made in an informed and engineer-like manner, saving both time and money [WS03].

SPE also enables complex load tests. These tests can cover usage scenarios, e.g., for highly dynamic cloud systems with worldwide deployment and multiple millions of users. To run such tests on a real installation can be nearly impossible, based on the load generation, substantial expenses, and not yet available hardware. With SPE, such tests can be realised for dozens of different design variants with lower costs [BDIS04].

Currently, there are two approaches that can be named as state-of-the-art approaches for model-based quality-of-service prediction and analysis: CloudSim[4] and Palladio[5]. While the former focuses especially on cloud applications and elasticity, the latter is a general-purpose approach, which works for all kinds of component-based systems. Due to this fact, we will focus in the following on the Palladio approach.

2.4.2.1. The Palladio Approach

Palladio is a model- and software component-based modelling approach that focuses on the prediction of quality attributes, and is therefore an example of a model-based analysis method on an architectural level [BKR09; RBH+16]. Palladio supports a variety of quality attributes, such as performance (i.e., response time) [BKR09], cost-efficiency [LE15], reliability [BKBR11], energy-efficiency [ÖGW+14], security [HFL16], and recently also scalability and elasticity [LB14]. Palladio uses its own DSL, which follows the example of UML. Therefore, it has a short adoption phase, and it is expected to have a high acceptance rate among software architects [BKR09].

[4] http://www.cloudbus.org/cloudsim/
[5] https://www.palladio-simulator.com/home/

To analyse an architectural design, the software architect has to specify a software and hardware model, as well as usage behaviour. Within the software model, the architect describes the behaviour, structure, and characteristics of the software. In the hardware model, the given hardware environment is described (for instance the HDDs, CPUs, and the system-landscape). Finally, the usage behaviour describes the behaviour of the user: How often a function is called, how many users are active at the same time, etc..

In the remainder of this section, we will continue explaining the details of PCM, describe standard solvers to analyse the architectural models, and introduce the AT extension, which will be used for contribution 1 (see Section 6).

2.4.2.2. PCM

Figure 2.19 gives an overview of the PCM and its elements. The PCM contains multiple main aspects, which are explained as follows:

Repository Diagram: In the repository diagram, the software architect models the components and their type. Further, he defines the required and provided interfaces of components here. Each component has a type, which is defined by (a) the provided interfaces of the type, and (b) by the required interfaces of the type. The syntax and semantics used in the diagram are similar to the UML2 Component Diagram [RQZ07].

Further, each component specifies a particular behaviour for each operation inherited from the provided interface. Within this behaviour specification, the software architect can model the behaviour of this operation, i.e., calling other operations or consuming resource demands, such as CPU or hard disk demands. In the PCM, the behaviour specification is called Service Effect Specification (SEFF). The SEFF is similar to a UML2 Activity Diagram; it can use, e.g., loops, branches, internal actions (to demand hardware resources like CPU cycles) and external actions (usage of other components that causes requires interfaces of the component).

System Diagram: In the system diagram, the components from the repository diagram are instantiated. The instances of components are called assembly context. Further, the system in the system diagram provides

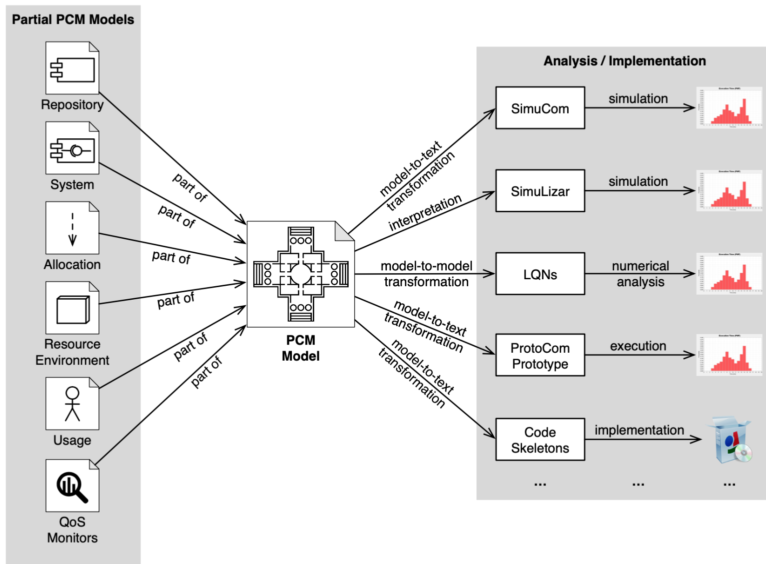


Figure 2.14.: Overview of the PCM (cf. [Leh18])

its interfaces, which represents the external access that is called from a user. These interfaces are forwarded to a provided interface of an assembly context. Also, a system can require interfaces, i.e., if an assembly requires external services (cf. [Leh18]).

Allocation Diagram: In the allocation diagram, it is specified which assembly (system) is allocated on which container (resource environment).

Resource Environment Diagram: In the resource environment diagram, the software architect models the hardware container on which the system is allocated. In the resource environment, it is possible to create multiple containers, which are interconnected via a network. Each container can represent a physical machine or a virtual server node. Each container can have an active resource, like CPUs or HDDs, for which the software architect needs to specify the processing rates and scheduling strategies (cf. [RBH+16]).

Usage Diagram: In the usage diagram, the software architect defines the user's behaviour. Here, the software architect can model different usage scenarios, chooseing from closed (fixed number of users) or open (users enter a specific arrival rate) workload models. In each scenario, individual user behaviour is modelled, and the user enters the system by the operations provided by the system.

QoS Monitor Diagram: In the QoS monitor diagram, it is specified which metrics are going to be measured and where during the analysis [Bec17]. Therefore, each entry in the QoS monitor points to a PCM element where the measurements should be taken, and the corresponding metrics which should be measured for that element (cf. [Leh18]). To give an example, in a QoS monitor, one can configure it to measure the response time of the system to a particular system operation of a system interface.

Each of the above models represents a part of a complete PCM model. The whole model can serve as input for different solvers described in the following.

2.4.2.3. Solver

To analyse a PCM model, a set of analytic or simulative solvers can be used (as shown in Figure 2.19). The result is a behaviour analysis of the complete system. This behaviour can be further analysed to identify limitations of the system, such as bottlenecks or SLOs violation. Afterwards, the model can be altered, and the consequences of the changes can be analysed. The analysis allows the SA to evaluate different versions of a system, before the first line of code is written.

Palladio offers a set of solvers, which we briefly characterise in the following. We will give more detail information about the solver needed for this thesis in the next section.

SimuCom: SimuCom is a simulation-based solver for the PCM. Its engine works based on a model-to-text (m2t) transformation, and, during the simulation, SimuCom can take measurements for a set of default metrics (i.e., response time).

SimuLizar: SimuLizar is the latest simulation-based solver for the PCM. SimuLizar interprets the PCM and provides measurements modelled in the QoS model (i.e., response time, utilisation, etc.). In contrast to SimuCom, SimuLizar can detect changes during the simulation in the instances. Due to this feature, SimuLizar can feature self-adaptive systems and enable reconfigurations during the simulation.

LQN: The Layered Queuing Network (LQN) is an analytical solver for the PCM. It is based on queuing networks, and it extends them by layers and elements, such as fork/join [KR08]. The LQN solver performs a model-to-model (m2m) transformation to create a LQN model of the PCM. Afterwards, the LQN models are solved with analytical and numerical mean-value approximation methods [KR08]. As a result, the LQN solver provides information in the form of, e.g., the mean response time of the system.

ProtoCom: ProtoCom is a Palladio extension that generates a runnable Java prototype out of the PCM. These prototypes hold the QoS constraints modelled in the PCM (e.g., resource demands), and can be executed in various target environments. With the help of prototypes, it is possible to run initial designs in real environments and evaluate the results concerning the SLOs.

CodeSkeleton: Besides the prototypes, Palladio supports m2t transformations to generate code skeletons from the PCM. A developer can use these code skeletons as a starting point for the implementation of the modelled system.

While analytical solvers are a lot faster in analysing the input model, they provide only information about mean values. Further, a simulation-based solvers offer more flexibility and freedom to the software architect, but can result in long simulation times, even for smaller systems.

For the understanding of this thesis, it is necessary to have a more detailed understanding of SimuCom (for Chapter 9), SimuLizar (for Chapter 8), and ProtoCom (for Chapters 7 and 9), which we give in the following.

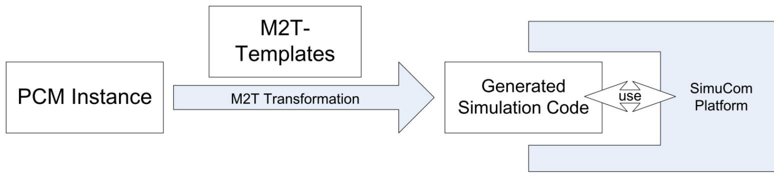


Figure 2.15.: Overview of the SimuCom Solver [Bec08]

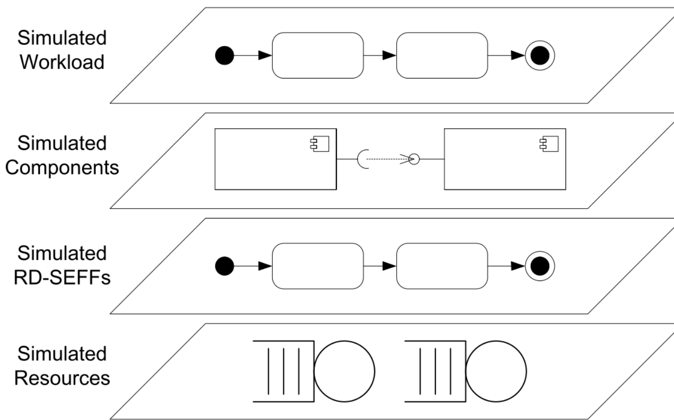


Figure 2.16.: Detailed View of SimuCom [Bec08]

2.4.2.4. SimuCom

Figure 2.15 shows the basic approach of the SimuCom solver. First, SimuCom takes as input a full PCM instance. Afterwards, it uses model-to-text transformations to generate the simulations code, which again is executed by the SimuCom Platform [Bec08]. The SimuCom Framework uses Discrete-Event-Simulation Modelling in Java (DESMO-J)[6].

To get a better understanding of the m2t transformation, Figure 2.15 gives a more detailed view.

[6] `http://desmoj.sourceforge.net/home.html`

The whole SimuCom simulation approach is based on the simulation of resources. Each resource is handled and simulated as a G/G/1 queue. The simulated workload component generates the load for the queues, and for each user, a thread is spawned that traverses through the simulated system [Bec08].

The simulation is based on a simulation of resources (see Figure 2.16). For this, SimuCom simulates the G/G/1. A simulated workload generates the load for the simulated resources. For each user, a thread is started which traverses the (simulated) system. When passing through the SEFF simulation, the resource demands in the form of stochastic expressions are evaluated to determine the resource demands. In general, there are two types of resources: The *CommunicationLinkResource* and the *ProcessingResources*. The latter are subdivided again into active resources (e.g., CPU or HDD demands) and passive resources (e.g., thread pools).

2.4.2.5. SimuLizar

SimuLizar is the next generation simulator and replaces the SimuCom simulator [BBM13; Bec17]. Therefor, SimuLizar is based on the SimuCom core framework as well. In addition to SimuCom, SimuLizar supports the analysis of self-adaptive systems, e.g., systems that scale dynamically depending on environmental factors, such as workload changes or service-level objectives violations. Further, SimuLizar gives more freedom when specifying the monitoring points. In contrast to SimuCom, the SimuLizar simulator does not generate simulation code. SimuLizar follows an interpreter-based approach instead. Meyer [Mey11] 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 interpreter-based approaches, the simulator traverses through the PCM instance and interprets the model elements it encounters. For the simulation logic, SimuLizar uses the core SimuCom framework. The simulation and interpretation process of SimuLizar contains two steps:

1. In the first step—the `SimulizarRuntimeState`— the setting up and configuration takes place. Thereby different model instances run the ModelObservers, in which, e.g., the `ResourceEnvironmentSyncer` is called, which creates `SimulatedResourceContainer` and

`SimulatedLinkResourceContainer` for each `ResourceContainer` or `NetworkLink` that is modelled inside the `resourceEnvironment` model. Next, the containers are stored in the resource registry of the SimuComModel.

2. In the second step, the simulation run, the PCM model interpreter traverses each user request and navigates through the various Palladio models. Thereby, the interpreter calls the correct interpretation for each model element. For example, first the user scenario model is interpreted, then all system calls in the user scenario are identified and interpreted. That way, the interpreter traverses through the models until it reaches the resource demands. Additionally, SimuLizer can consider self-adaptive behaviour [Bec17].

2.4.2.6. ProtoCom

Like the above two solvers, ProtoCom is also a Palladio analyser. A common method of design evaluation is performance prototyping. For this purpose, ProtoCom offers a method for generating runnable Java applications from the PCM instances. Thereby, it uses model-to-code (m2c) transformations [KL14]. These applications can be run in realistic environments, and the software developer can check the monitoring data against the SLOs.

ProtoCom Transformation The process of the m2c transformation is shown in figure 2.17. The input of the transformation is a PCM instance. The transformation generates a runnable performance prototype. The prototype consists of the generated code and the ProtoCom framework.

During the m2c transformation, ProtoCom traverses through the PCM instances, and transforms the processing resource demands into synthetic resource demands (e.g., calculating Fibonacci)[7].

To match the specified resource demands in the model, ProtoCom needs to run a calibration on the target platform. The calibration step is required only once. Afterwards, the target platform is no longer needed. Moreover, it is possible to run all experiments on a host machine.

[7] A full description of all available demands is given in Chapter 5

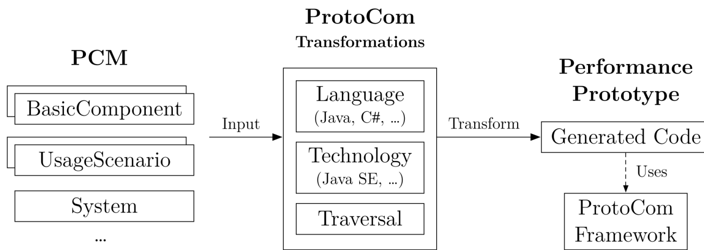


Figure 2.17.: Overview of the ProtoCom M2C Transformation [KL14]

Java SE RMI Prediction Prototype ProtoCom can provide various target applications (like Java SE or Java EE). In the following, we have a closer look at the Java SE RMI prediction prototype. This will become most relevant in Chapter 9.

Figure 2.18 shows the architectural view of a JavaSE performance prototype. As one can see, the prototype consists of two parts: first, the prototype (above the dotted line) and second, the ProtoCom framework (below the dotted line). The latter is the same for each prototype and contains the ProtoCom logic. The prototype varies and reflects the PCM input instances directly.

Especially interesting for us is the `AbstractResourceEnviroment`. This component contains all the different resource demands. By default, ProtoCom uses a Fibonacci demand to represent the load on the CPU (for CPU-intensive load). However, other demands, such as sorting array demand (for I/O-intensive tasks), are available.

Resource Demand Mapping Given the work from Becker [Bec08], there are two ways to map independent resource demands to hardware-dependent ones.

The first approach involves the introduction of a constant scaling factor. This requires the knowledge of the hardware‘s capabilities. For example, one work unit could correspond to the calculation of 100, 000 Fibonacci numbers. However, the knowledge of this factor and the accuracy of this approach is highly questionable.

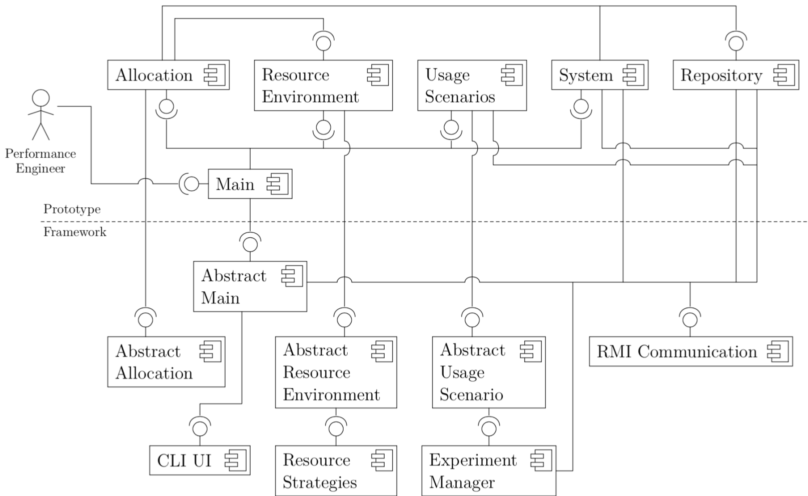


Figure 2.18.: Architectural View of JavaSE Performance Prototype [KL14]

Therefore, the second approach is based on an automated performance detection of this factor. Thereby, a benchmark is run on the target machine to determine the factor. The output of this benchmark is a calibration table. This table includes two columns: the first column shows the time in *ms* and the seconds, the input parameter for the Fibonacci function (e.g., how many numbers should be calculated).

We will explain the resource demands and the approach behind it in more detail in Chapter 5. For more information on the ProtoCom approach, we refer to [Bec08].

2.4.2.7. Architectural Templates

S. Lehrig proposed the AT approach in [Leh18] to enable software architects to easily reuse architectural knowledge in the form of reusable AT in the context of architectural analysis. Lehrig included a proof of concept of the AT method in Palladio. We will use the AT method in Chapter 6 to build a parallel architectural catalogue. Therefore, we will explain the details of the

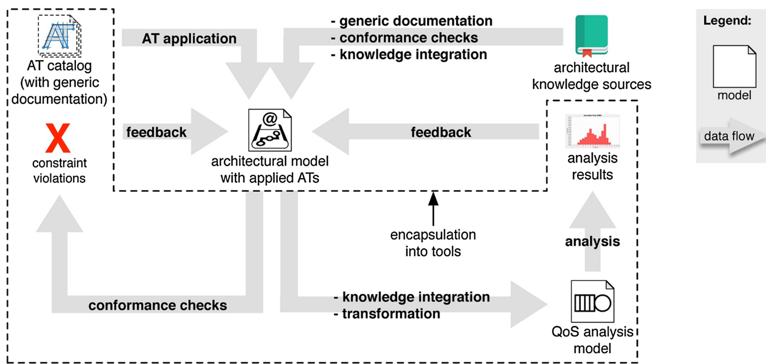


Figure 2.19.: Process of the AT Application Process (cf. [LHB18])

AT method in the course of this section. To follow Chapter 6, it is necessary to understand the basics explained in the following.

The Architectural Template Method

> "The AT method is a software engineering method with which software architects can reuse architectural knowledge from pre-specified templates—ATs—for architectural modelling and architectural analyses. AT engineers specify the AT, that is, implemented, quality-assured, and provided within catalogues. In applying ATs from such catalogues, software architects become more effective and efficient in their architectural analysis tasks." [LHB18]

The AT method differs between two views: (a) the view of the software architect who wants to use ATs, and (b) the AT engineer who creates the ATs. In the following, we will explain the use of ATs and their creation in detail as proposed by Lehrig [Leh18; LHB18].

Usage of an Architectural Template To use an AT, the software architect needs first to model the software architecture of the desired system. During this process, the software architect can choose and apply suitable ATs from the provided AT catalogue. The catalogue provides different QoS specific

templates. For example, a load balancer template can improve response time. To use the AT, the software architect binds the corresponding roles to the software architecture, and configures the parameters of the AT. When using the AT tool, it prevents misconfiguration or violation of AT constraints (e.g., having illegal connections between model elements). Before analysing the model by the solver, the AT engine performs a m2m transformation and weaves AT completions into the architectural model (e.g., a load balancer [Leh18]).

Creation of an Architectural Template The AT engineer creates AT templates and provides them via an AT catalogue to the software architect. An AT catalogue contains ATs for a specific topic, like architectural styles or parallel architectural patterns.

The first step in the creation of a new AT is the identification of need and the corresponding QoS properties (e.g., response time) and which metrics need to be measured, as well as a suitable analysis approach (e.g., Palladio). In the next step, the AT engineers need to gather and extract the reusable architectural knowledge, and to formalise it within an AT. Thereby, the AT engineer needs to specify roles, completions, and constraints, and bind first-named to architectural elements. In the last step, AT engineers ensure the quality and correctness of the AT, e.g., by testing.

2.5. Hierarchical Queueing Petri Nets

Especially for the first contribution of this thesis (see Chapter 6), we are using Hierarchical Queuing Petri Net (HQPN) to formally describe the dynamic behaviour of the parallel language elements in the PCM. Therefore, we will briefly describe the foundations of HQPN here.

HQPNs include several extensions to the conventional Petri Net (PN)s. These extensions include the Coloured Petri Net (CPN), Generalised Stochastic Petri Net (GSPN), Coloured Generalised Stochastic Petri Net (CGSPN), and QPN. In the following, we assume the reader is familiar with PNs. Therefore, we only give a brief introduction to PNs and HQPNs. Thereby, we follow

the definitions given by [BK02] for PNs and the various extensions [Jen13]. A more detailed overview is provided in [Koz08].

2.5.1. Petri Nets

An ordinary PN is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$, where:

1. $P = p1, p2, ..., p_n$ a finite and nonempty set of places;
2. $T = t1, t2, ..., t_m$ a finite and nonempty set of transitions $P \cap T = \emptyset$;
3. I^- and $I^+ : P \times T \longrightarrow \mathbb{N}_0$ are called backward and forward incidence functions, respectively;
4. $M_0 : P \longrightarrow \mathbb{N}_0$ is called initial marking.

PNs cannot differ between the token type. A CPN allows the user to bind a type (colour) to each token. Each place is restricted to a set of colours. Furthermore, the transitions of CPNs can fire in different modes based on the colour of the token.

In addition, using Stochastic Petri Net (SPN)s, we can include temporal aspects. SPN assigns an exponentially distributed firing delay to each transition. This delay defines the time a transition waits after being enabled until it fires [Koz08].

2.5.2. Queuing Petri Nets

Bause et. al [BK02] introduced QPNs. QPNs are based on CGSPNs and integrates the queue concepts into places. Therefore, QPNs are used to express queuing behaviour, which are in the form of SPE, in PNs. In QPN, there is a queueing place, where tokens are queued, and a depository for tokens which have completed their service.

Models in QPN can become quite large. To tolerate the size problem of monolithic QPNs, it is convenient to divide them into smaller inter-active subnets. For this purpose, HQPNs are used. They consist of several QPNs subnets and additionally contain subnet places. Each subnet has a dedicated input and output place, as well as another place counting the active population of

the subnet, which is the number of tokens fired into the subnet that have not yet left the subnet.

According to Bause et. al [BK02] a Hierarchical Queueing PN is a 4-tuple $HQPN = (N, SP, SA, FS)$, where:

1. N is a finite set, where:

   a) $n \in N$ is a non-hierarchical QPN $\left(P_n, T_n, C_n, I_n^-, I_n^+, M_{n_0}, Q_n, W_n\right)$,

   b) the sets of net elements are pairwise disjoint: $\forall n_1, n_2 \in N : \left[n_1 \neq n_2 \Rightarrow \left(P_{n_1} \cup T_{n_1}\right) \cap \left(P_{n_2} \cup T_{n_2}\right) = \emptyset\right]$

2. $SP \subset P_N$ is the set of the subnet places,

3. $SA : SP \rightarrow N$ is the subnet assignment function,

4. $FS \subseteq \mathcal{P}\left(P_N\right)$ is the set of fusion sets, such that members of a fusion set have identical colour sets and equivalent initialisation expressions: $\forall fs \in FS : \forall p_1, p_2 \in fs : \left[C\left(p_1\right) = C\left(p_2\right) \wedge M_0\left(p_1\right) = M_0\left(p_2\right)\right]$

2.6. Summary of Foundations

In this chapter, we presented the foundations needed to follow the course of the thesis. Since not all foundations are necessary to understand certain contributions, Figure 2.20 provides an overview of the contributions, and of the sections of the foundations required to follow.

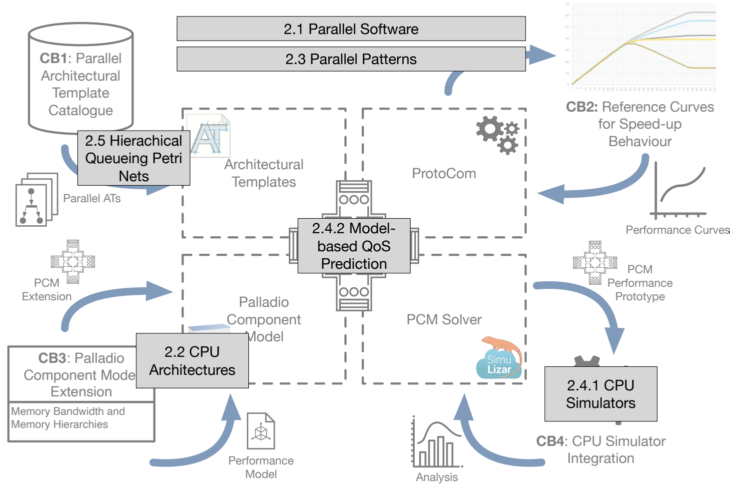


Figure 2.20.: Mapping of Foundations to Contributions

In the next chapter, we will continue with outlining the research design.

3. Research Design

This section introduces the research design followed in this thesis. It clarifies the overall research goal, the research questions to be answered in the course of this thesis, and the process followed to answer the questions.

In model-based QoS prediction, the highest goal is to be as precise as possible about the predictions of the desired quality attribute in comparison to the real system (accuracy).

Since the focus of this thesis is performance prediction, we will look only at the quality attribute performance. The current state-of-the-art model-based performance prediction approaches focus only on a single metric—CPU Speed—when specifying the characteristics of processor architectures. Single-metric models might be fine for most single-core architectures. However, recent experiments have shown that current performance models produce insufficiently accurate predictions when analysing parallel applications in multicore environments [FH16; FSH17]. Therefore, we formulate the following hypothesis, on which this thesis is based:

Hypothesis 1 ($H_{0.1}$):
There exist additional CPU architecture and memory hierarchy related performance-influencing factors—besides CPU speed—which have an impact on the performance of parallel application.

Hypothesis 2 ($H_{0.2}$):
When considering the additional performance-influencing factors in performance prediction models in an abstract form, in architectural models, and during design time, we can improve the accuracy of the model-based performance predictions for parallel application.

Validating the hypothesis will result in a set of emerging research questions: How can parallel applications be modelled; how far off are current predictions; what are the relevant performance-influencing factors; do other approaches exist to predict the performance of parallel applications; etc.

Since model-based performance prediction is often used during the early design phase, it is important that predictions based on abstract software architectures are reliable—also for parallel applications—to ensure a high level of quality and to foster the use of engineering-like approaches. Therefore the overall goal is defined as follows:

> **Research Goal (*RG*):**
> Improving the accuracy, usability, and applicability of model-based QoS predictions of the performance of parallel applications in multicore environments.

3.1. Research Method

To achieve its goal, this thesis follows the the design science approach in combination with the method for experiment-based performance model derivation proposed by Jens Happe [Hap08]. According to this method, the performance model is extended in steps. First, a minimum set of additional attributes are identified in a goal-oriented manner. Second, the additional attributes are added to the performance model. Third, the performance is evaluated and checked to see if it meets the requirements. If so, the model derivation terminates. If not, further performance attributes are identified, and steps two and three are repeated. The evaluation—checking whether the requirements are met—is based on an experiment validation. One chooses a concrete scenario, sets up an experimental environment, and uses the experiment's results to evaluate the altered performance model [Hap08].

In contrast to behavioural science, whose goal is truth, the outcome of this thesis is one or multiple useful artefacts. Therefore, the design science approach, whose goal is a utility (cf. [HC10]), is most suitable and is applied in the course of this thesis. Figure 3.1 shows the design science approach we have chosen. The core artefact—in the middle of the figure—is the improved

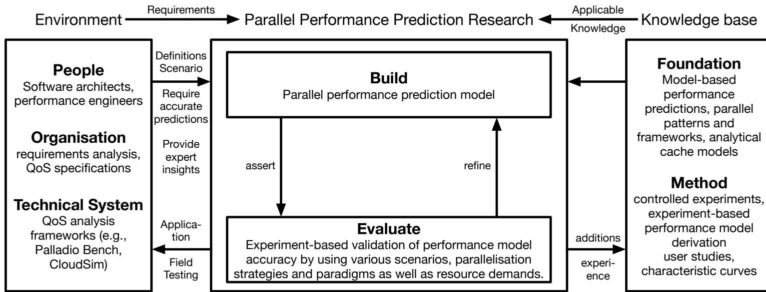


Figure 3.1.: Applied Design Science Framework to Archive the *RG* ([HC10])

performance prediction prototype, including improved prediction models, enhanced tooling, and adjusted processes. We evaluate this artefact using the experiment-based performance model derivation method, mentioned above. The environment provides the requirements for the artefact, particularly the requirements of software architects and performance engineers, who have a real-world need for accurate parallel performance predictions, and therefore, also for the evaluation. The environment also defines the use case scenarios and provides further insights from expert interviews.

The insights gained during the evaluation of the artefacts can not only be used to improve the artefacts further, but can also add to the knowledge base. Vice versa, the artefact builds and is improved by the current state-of-the-art techniques, methods and knowledge. As a last step, the environment is used to conduct a field test and confirm the quality of the artefact in production or semi-productive environments.

To find the relevant metrics for the evaluation, to break down the overall *RG*, and to reveal additional contributions, the formulation of research questions helps. In the following, the research questions for this thesis are introduced and explained based on the thesis process overview given in Figure 3.2.

3.2. Research Questions

Given the research method described above, a concrete process can be extracted. In this section, we will describe the research process (see Figure 3.2) step by step. While doing so, we will break down the overall hypothesis $H_{0.1}$, $H_{0.2}$, and RG into smaller and easier-to-evaluate research questions and assign them to the process steps. In so doing, we identify four main questions, which we break down into subquestions. For each question, we will give a detailed explanation as well as introducing the hypothesis on which we have based the research question.

The first step shown in Figure 3.2, is to verify or falsify the base hypothesises $H_{0.1}$ and $H_{0.2}$, and to identify the research need. We verified the hypothesis in [FH16; FSH17], where we performed a scenario evaluation of the capability of a state-of-the-art performance prediction approach (Palladio). For this, we used two different parallelisation paradigms—Java threads and AKKA Actors—to implement and parallelise two standard parallel applications, namely a matrix multiplication and a bank transaction scenario. Further, the scenarios were modelled and analysed with Palladio. Finally, we compared the Palladio analysis results with the measured execution times of the applications. Simply put, the results show that the predictions are off by up to 63%.

The next logical step is to perform a SLR to identify all related research in the field and to discover possible solution strategies unknown to our community. The SLR is described in Chapter 4.

Further, the lessons we learned from the experiment led us to hypothesise H_1 to H_4, explained next.

3.2.1. RQ_1: Performance Modelling of Parallel Behaviour

Software Behaviour: When we talk about software behaviour in the following, we relate to the performance influencing aspects of the behaviour. Thus, we model abstract elements of the control flow of the software and the path the application takes through the program. The model's pragmatism is based on the idea to reflect the program's performance as good as possible concerning wall clock time. We do not focus on

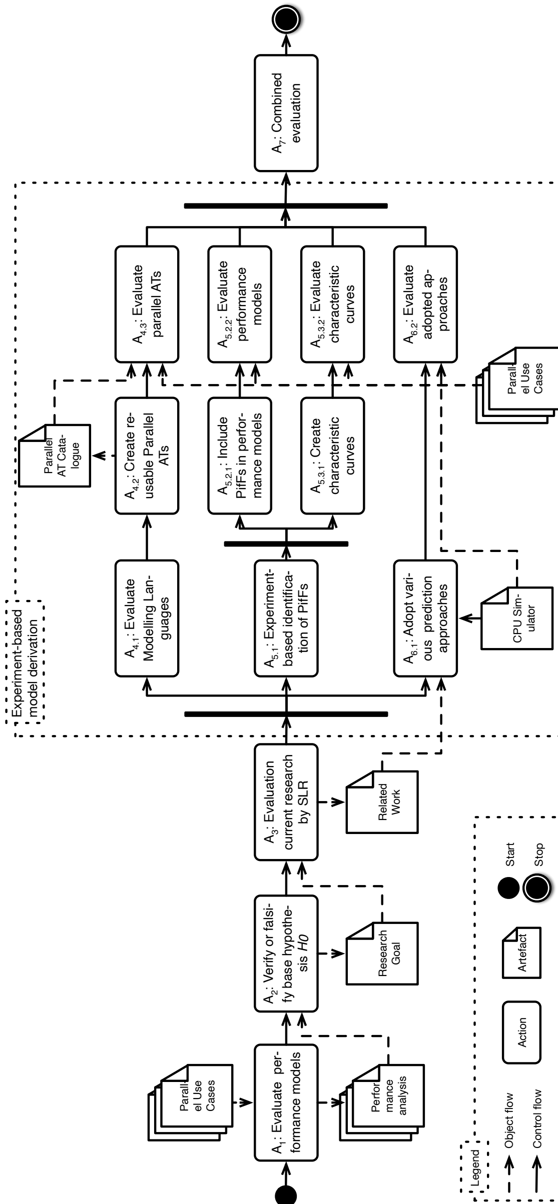


Figure 3.2.: Research Process of this Thesis

the formal semantics of the behaviour. Moreover, we relate to the performance characteristics and the performance relevant demands a software behaviour executes on its hardware, especially in multi-core environments. Relevant aspects are (but not exclusively) forking and synchronising of threads, data read and write operations, and resource-demanding operations.

Keeping that in mind, our first hypothesis H_1 relates to the abilities of current modelling languages to represent the needs and performance characteristics of parallel software behaviour. Often parallel behaviours do the same task, but with different data (e.g., parallel loops—Section 2.3—or SIMD—Section 2.2). Therefore, modelling the same behaviour over and over again is time-consuming, error-prone, and simply not possible for highly parallel systems.

So, to verify or falsify H_1 we raised $RQ_{1.1}$ and $RQ_{1.2}$. Further, $RQ_{1.3}$ was defined to answer the question of how to improve modelling languages if H_1 is verified.

These research questions relate to the actions $A_{4.1}$ to $A_{4.3}$ in Figure 3.2, where first the current modelling languages are evaluated; next, an extension in the form of a parallel AT catalogue is created; and last, the extension is evaluated based on a user study to prove its effectiveness.

RQ_1: **Modelling of parallel performance relevant behaviour in massive parallel environments**:

H_1: *Current modelling languages (e.g. UML) have only limited expression power and are insufficient to express the performance relevant behaviour of highly parallel software.*

$RQ_{1.1}$: Are software architects able to model even simple parallel concepts of highly parallel systems in an efficient way? Thereby, SA needs to focus on abstract performance relevant attributes on architectural level during early design time.

$RQ_{1.2}$: Are software architects able to model the parallel software behaviour of an application with the help of current modelling languages, so that (a) the relevant performance characteristics

are captured and expressed, and (b) all necessary information for performance evaluation is covered?

$RQ_{1.3}$: How can software architects be supported in the task of creating accurate performance prediction models efficiently?

3.2.2. RQ_2: Behaviour of Highly Parallel Applications

The second research question focuses on the performance behaviours characteristics of highly parallel systems in parallel environments (multicore architectures). The assumption here is that the selected parallelisation paradigm, as well as the architecture characteristics, have a high impact on the performance of an application and therefore need to be considered in the performance predictions ($H_{2.1}$).

The $RQ_{2.1}$ therefore focuses on observing the parallel application execution, while the $RQ_{2.2}$ aims to identify the most relevant performance-influencing factors (Action $A_{5.1}$). $RQ_{2.3}$ covers the observation from [FSH17], in which we noticed that the selection of the parallelisation paradigm may have an impact. A validation of this hypothesis ($H_{2.1}$) is needed. Finally, $RQ_{2.4}$ aims to identify common characteristics in the execution of parallel behaviours, which can be described in characteristic curves (Action $A_{5.3.1}$). These curves can be included in the model predictions to increase accuracy.

RQ_2: **Performance behaviour of highly parallel applications in massive parallel environments**:

$H_{2.1}$: *The speedup and performance behaviour of highly parallel applications depends heavily on the chosen parallelisation strategy or paradigm.*

$H_{2.2}$: *The hardware architecture (e.g., number of CPU cores, memory bandwidth, memory hierarchies) of the execution environment has a strong impact on the performance of the parallel applications.*

$H_{2.3}$: *The speedup of a parallel application is not only influenced by the number of cores available in a system but also by additional hardware specific performance-influencing factors.*

$RQ_{2.1}$: How do highly parallel applications behave in massive parallel environments (multicore systems) regarding response time (speedup), memory access rates (L1, L2, L3, Random Access Memory (RAM) usage), and memory bandwidth utilisation?

$RQ_{2.2}$: What factors influence performance the most in highly parallel applications?

$RQ_{2.3}$: Does the choice of parallelisation strategy have a significant impact on behaviour?

$RQ_{2.4}$: Do highly parallel applications show similar behaviour, which can be described by one or multiple performance curves?

3.2.3. RQ_3: Performance Prediction Models

This research question deals with performance prediction models for parallel applications. H_3 is the baseline hypothesis here, and $RQ_{3.1}$ is designed to verify that.

Based on $RQ_{2.2}$, $RQ_{3.2}$ aims to answer the question of which performance-influencing factors need to be included in the prediction model (Action $A_{5.2.1}$). At the same time, $RQ_{3.3}$ covers the evaluation of the altered performance prediction models (Action $A_{5.2.2}$).

RQ_3: **Performance prediction models**:

H_3: *Current model-based performance prediction models fail to consider relevant performance-influencing factors for parallel systems and thus their predictions are off.*

$RQ_{3.1}$**:** Are current simulation-based performance prediction approaches capable of predicting the performance of parallel and highly parallel systems accurately?

$RQ_{3.2}$**:** If not, what are the missing characteristics of software behaviour that must be included in performance prediction models (performance-influencing factors)?

$RQ_{3.3}$**:** Can modelling the additional performance-influencing factors improve the overall accuracy of performance prediction?

3.2.4. RQ_4: CPU Simulators

As explained in Section 2.4.1, CPU simulators can simulate the behaviour of parallel applications in multicore environments based on a given implementation. Therefore, the hypothesis here is that these CPU simulators, included in the performance prediction process, can help improve the quality of prediction (H_4). The significant challenge here will be to find suitable simulators that work with architectural designs ($RQ_{4.1}$) and integrate them into the existing approaches and tooling ($RQ_{4.2}$ and Acton $A_{6.1}$). Finally, $RQ_{4.3}$ evaluates the quality of the integrated approach (Action $A_{6.2}$).

RQ_4**: CPU simulators for architectural performance predictions**:

H_4**:** *CPU simulators—used in other domains (e.g, hardware vendors)—can help to improve predictions for parallel applications on multicore CPUs.*

$RQ_{4.1}$ Can CPU Simulators be used by software architects to evaluate the response time of parallel architectural designs?

$RQ_{4.2}$ How would the integration of CPU simulators alter the process of performance predictions?

$RQ_{4.3}$ Does the use of CPU Simulators increase the performance prediction accuracy for parallel applications in multicore environments?

3.3. Research Design Evaluation

Addressing the RQ satisfactorily, and providing an artefact that is beneficial for the given use cases, is essential for a design science approach [HC10]. Therefore, we will lay out the evaluation of the contributions in this section and follow the concepts pointed out by [SV12]. Sonnenberg and vom Brocke argue for a continuous evaluation of artefacts and sub-artefacts throughout the whole research project. As Figure 3.2 shows in action $A_{4.3}$, $A_{5.2.2}$, $A_{5.3.2}$, and $A_{6.2}$, each research question (contribution) is evaluated separately. While in $A_{5.2.2}$, $A_{5.3.2}$, and $A_{6.2}$ the artefacts are compared against the current state-of-the-art approaches, $A_{4.3}$ is evaluated by a user study to prove the usability of the artefact empirically. After this, an individual evaluation and additional integrated evaluation of the combined artefacts is planned (Action A_7). Further details of the specific evaluation methods are provided in the corresponding chapters of the contributions.

3.4. Design Science Guidelines

To perform an adequate design science experiment, Hevner et al. provide seven guidelines, which they recommend addressing in a project-specific manner [HC10]. The guidelines are listed below, and we briefly describe how we have addressed them in this thesis:

Gl_1 **Design as an artefact:** This thesis will result in multiple artefacts. First, it provides a modelling language extension to specify parallel behaviour within models (see Chapter 6). Second, it provides a model or model extension that captures the relevant characteristics of multicore architectures (see Chapter 8), which can be used for performance predictions. Third,

it provides a model that captures the characteristic speedup behaviour of highly parallel applications, including the relevant performance-influencing factors (see Chapter 7). This can be used to estimate the maximum speedup of applications. Fourth, it provides a method to include CPU simulators in the process of predicting performance (see Chapter 9) to get very accurate parallel behaviour predictions.

Gl_2 **Problem relevance:** As [FH16; FSH17] showed, the need for accurate performance predictions is highly relevant, as current prediction models are far off. Moreover, in the papers, they only considered a medium parallel multicore system with 16 cores. The current state of the art is already 32 to 64 cores for desktop PCs.

Gl_3 **Design evaluation:** The utility, quality, and efficacy of the design artefacts is rigorously demonstrated by use case evaluations and commitment to state-of-the-art performance. If an artefact does not perform with better accuracy than the current state of the art, the artefact is considered depraved.

Gl_4 **Research contributions:** The main contribution of this work is an improved performance prediction for parallel applications in multicore environments. An added benefit is its contribution to the knowledge base.

Gl_5 **Research rigour:** Strict, rigorous, and peer-reviewed methods are used to achieve the research goal, e.g., SLRs, the experiment-based performance model derivation proposed [Hap08], and guidelines for user studies and experiment evaluations (e.g., GQM).

Gl_6 **Design as a search process:** It is necessary to satisfy existing laws and best practices in the application area of the artefacts domain. Identifying laws and best practices is achieved by a SLR covering this and neighbouring domains, as well as by expert interviews from academia and industry.

Gl_7 **Communication of research:** The results are transmitted to both industry and the academy, who will both benefit from this information, by means of various peer-reviewed conference and workshop papers. The publications are summarised in Appendix A.1.

Given that, we will continue in the next chapter with A_3 and research and describe the related work.

4. Related Work

Following the research design described above, we came to ask ourselves if existing research had face challenges similar to what we faced in [FH16; FSH17]. To answer this question as extensively as possible, we decided to perform a full SLR according to Kitchenham [KBB+09]. A SLR has two advantages: First, we may find useful approaches that we can use to overcome our challenges. Second, at the same time, we delineate the research area and cover related work.

In this section, we elaborate on step (A_3) in the research process and present the SLR design and results. This SLR was sucessfully peer-reviewed and published in [FHLB17]. For the sake of being up-to-date, we re-executed the SLR for this thesis and added the delta of resources found. This ensures that we focus only on research question $\mathbf{R}_{SLR-2}$ (see Section 4.2.1), which is especially relevant for this thesis.

4.1. SLR Overview

Even though performing a SLR comes with additional overhead, it also brings a set of advantages. First, Kitchenham [KBB+09; KDJ04] provides a detailed reference process to follow step by step. Second, if the review protocol (search method) is well designed, the outcome of the search is reproducible and more importantly, scientifically elaborated and reusable.

Figure 4.1 shows the process we followed during the SLR. We split the whole process into three phases: Planning, conducting, reporting. In brackets, we indicate how many sources remain for further processing. The first number (red) shows the sources from the 2016 run, and the second number (blue) from the 2020 re-execution.

In the course of the chapter, we elaborate on each phase and each step in detail. Finally, we give a conclusion and an overview of the related work.

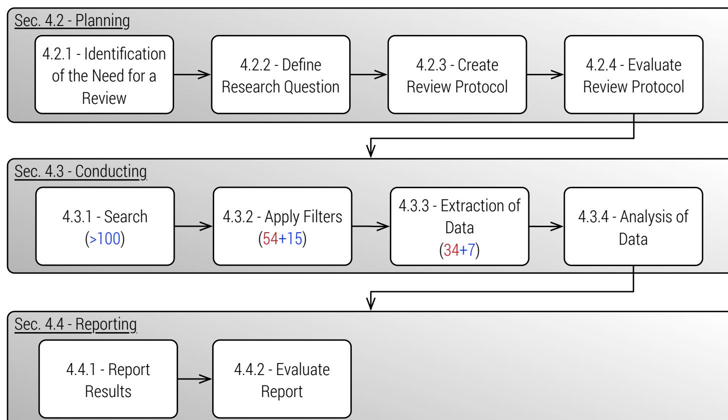


Figure 4.1.: Overview of the Systematic Literature Review Process (cf. [KC07])

4.2. SLR Planning

The first phase, the planning phase, will result in the review protocol, which is the most important artefact of the SLR. It defines the whole process, containing the search strategy, inclusion and exclusion criteria, and the data extraction process. Further, we define the search goal and research questions here. In the following, we report on each step in detail.

4.2.1. Research Questions

As we have already elaborated on the need for research [FH16; FSH17], we will skip this step and start on the research question, which sets the primary direction of the search.

Given our domain, we focus on software developers and architects. We search for modelling approaches that enable software architects to analyse

and predict the performance of parallel software in multicore environments during the design phase. Thus, we aim to answer two concrete research questions [FKB18]:

R$_{SLR-1}$ Which modelling approaches exist for performance prediction in different parallel programming paradigms, and what are their practical uses?

R$_{SLR-2}$ Which concepts exist to predict the performance of parallel software in multicore environments?

4.2.2. Review Protocol

Given the research question, we create the review protocol, which is the central artefact created during the first phase. All further steps are aligned to the definitions established in the review protocol. Thus, its quality is crucial for the SLR. To guarantee high quality, we develop the review protocol iteratively. At the end of each iteration, we validate the review protocol against a set of sources that we want to ensure are included, and a set of sources that we want to ensure are excluded. We pick theses sources manually upfront.

For the sake of simplicity, we describe only the final version here.

4.2.2.1. Search Strategy

In the search strategy, we define which search engines we use and how we construct the search terms to create the search phrases.

Our first decision here is to use Google Scholar[1]. Using Google Scholar is suggested by Kitchenham [KC07] because Google Scholar is a meta-search engine and includes most sources of scientific publications—also from other relevant databases.

Next, we derive the search terms. The initial set of search terms we gain from our knowledge and the already-known related work. During the iterations,

[1] `https://scholar.google.com/`

we update the collection of search terms, based on the results, and also include synonyms.

The final list contains the following search terms—synonyms not listed: *Parallel Programming*, *Many Core*, *Multicore*, *Modeling*, and *Software Performance Engineering*. To get the search phrases we combine these terms using "*AND*"- and "*OR*"-operators. This leads to the following search terms [FHLB17]:

T_1: *("Parallel Programming") AND ("Modeling")*

T_2: *("Multicore") AND ("Modeling" OR "Software Performance Engineering")*

T_3: *("Multicore") AND ("Parallel Programming") AND ("Modeling" OR "Software Performance Engineering")*

We also use our synonym list to replace keywords by synonyms. This way, we can cover a more extensive range. An example, based on T_3 and synonyms is:

$T_{3_{mod1}}$: *("Many Core") AND ("Performance Modeling" OR "Software Design") AND ("ACTORS")*

Further, we created a blacklist with terms we expected to come up in our search that are outside our specific domain. For example, we blacklisted *weather*, since we expected to find sources focused on weather prediction models. The full list of synonyms, blacklist, and keywords, along with all results, are available online[2].

4.2.2.2. In- and Exclusion Criteria

After agreeing on the search strategy, we define in- and exclusion criteria to filter identified sources and to focus on relevant documents. In our case, we consider all sources that fulfil one of the following statements [FHLB17]:

[2] `https://doi.org/10.5281/zenodo.3972806`

Inclusion Criteria

I_1: Sources that describe a modelling language or a language extension for modelling parallel programs.

I_2: Sources that address the limitation or potentials of existing modelling languages for parallelism.

I_3: Sources that give details or definitions of one of the search terms.

I_4: Sources that talk about techniques for paradigms of multicore systems.

I_5: Sources that give performance prediction models for multicore systems.

Additionally, we define the following exclusion criteria. We will not consider sources that fulfil one of the following criteria [FHLB17]:

Exclusion Criteria

E_1: Sources focusing on prediction models for subjects other than software (e.g., weather prediction models) and where the general topic is not computer science or programming.

E_2: Sources that do not address problems with parallel programming or performance prediction.

E_3: Sources that do not include the search terms in the title or abstract.

E_4: Panel discussions, prefaces, tutorials, book reviews, or presentation slides; we prefer to focus on genuine publications.

E_5: Sources in languages other than German or English.

E_6: Sources that are inaccessible through public or the university access from the TU Chemnitz or Uni Stuttgart.

E_7: Sources published before 2003: Because it was only around 2003 that multicores became common in desktop computers. Also, software performance engineering was not commonly known before.

E_{new1}: We will not consider our own sources during the repetition of the SLR.

To decide whether to consider a source or not, we first apply the inclusion criteria. Next, we apply the exclusion criteria to all sources passing the inclusion criteria check. If a source fulfils the exclusion criteria, we eliminate it from consideration.

4.2.2.3. Quality Indicators:

The next step is to evaluate the remaining sources. For this, we use quality indicators. In the following, we will introduce the quality indicators. To pass the evaluation step, a source has to at least partly fulfil at least one quality indicator:

Q$_{1a}$ Does the source address problems with parallel programming?

Q$_{1b}$ Does the source identify problems or open questions?

Q$_{2}$ Does the source provide techniques, paradigms, or patterns to apply parallelism to software?

Q$_{3a}$ Does the source introduce a modelling approach to deal with the complexity of parallelism?

Q$_{3b}$ Does the source evaluate a modelling approach that deals with the complexity of parallelism?

Q$_{3c}$ Does the source introduce or evaluate an approach to predict quality attributes of parallel software (in multicore environments)?

4.2.2.4. Data Extraction

Next, we need to define how the data is extracted from the remaining sources. For this, we define a three-step process. First, we collect bibliographic information about the source (e.g., authors and date of publication). Based on this information, along with the absolute number of keyword hits in the title, we rank the sources. Second, we extract and summarise the sources by evaluating the abstract, introduction, and conclusion (in order of ranking). In the process, we re-evaluate the in- and exclusion criteria. Third, we perform a full paper review for the remaining papers. During the review, we again double check in- and exclusion criteria.

Characteristic	Values
Domain	General Software Engineering, HPC, Embedded Systems, Software Performance Engineering (SPE)
Source Type	Problem Statement, Solution Introduction, Experience Report, Knowledge Accumulation
Pattern Type	Design Patterns, Programming Patterns, Architectural Patterns, Not Available
Technique	Modeling Paradigm, Programming Language, Library/Framework, Not Available

Table 4.1.: Characteristics Used for Categorising [FHLB17]

4.2.2.5. Data Analysis

After data extraction, we evaluate and interpret the extracted data. We categorise the data using the four characteristics shown in Table 4.1.

The first dimension of categorisation is the domain. Here we distinguish between sources contributing to the domains of Embedded Systems, HPC, or SPE. Sources that target software engineering, in general, are assigned to General Software Engineering.

The second dimension is the source type: Problem Statements focus on open issues; Solution Introductions provide an approach; Experience Reports describe practical realisations (e.g., case studies); and Knowledge Accumulations summarise a wide field of knowledge (e.g., surveys).

We expect numerous sources to target parallelisation patterns or techniques. Thus, the third and fourth dimension splits these source groups according to the pattern or technique each focuses on. In case no pattern or technique is described, we tag it as *Not Available*.

4.2.3. Evaluate Review Protocol

To evaluate the review protocol, we execute two evaluations. First, as mentioned, we perform multiple iterations. In each iteration we execute a small test search and check that pre-defined sources are included or excluded correctly.

Second, we form a review board within our group, including experts from SPE, Model-driven Software Development (MDSD), and HPC domains, which are the most relevant domains for our search. Within this board, we review each iteration run. In total, we had three iterations within the full group and several discussions in groups of two.

4.3. SLR Conducting

Once the SLR is planned, the implementation phase begins (see Figure 4.1). In this section, we describe how to perform the SLR as defined in the review protocol (Section 4.2.2). We perform the actual search, apply the filters to the sources found, and analyse the data retrieved. For the sake of simplicity, we give only a summary of the results. The complete raw data and documentation are available in our repository[3].

4.3.1. Executing the Search

To evaluate our search phrases and terms, we performed test searches with strict automatic filtering based on our blacklist rules. Due to the small number of results (three), we relaxed the blacklist rule by removing the word *weather forecast*, which led to the expected result that the sources found covered a more comprehensive range. Therefore, we decided to manually preselect sources based on title only, evaluating the title of the sources one by one. Only those sources that passed the evaluation were considered in further steps. On December 11, 2016, we conducted the search of the first run and obtained 54 sources after the manual pre-selection. On June 14, 2020, we reran the SLR. We focused only on $\mathbf{R}_{SLR-2}$ and executed only the query $\mathbf{T}_3$ with its variations. We received a delta of 15 new papers.

4.3.2. Applying the Filters

With the initial result set at hand, we apply our filter criteria step by step. As mentioned above, we performed the first evaluation during the search.

[3] https://doi.org/10.5281/zenodo.3972806

We manually evaluated all sources based on the title. To minimise personal bias, we ensured that only sources from other areas were excluded and only if the title provided sufficient evidence for exclusion. Figure 4.2 shows the filtering process and the number of sources after each step.

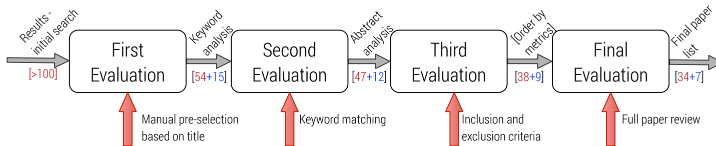


Figure 4.2.: Filtering Process

During the second evaluation, we check that the search terms are mentioned in the title, the keyword section, or at least in the abstract. All sources not mentioning at least one of our search terms were excluded. We ended up with 47 sources after this step. The rejected sources only mentioned the keyword in the full text, where we assume that it had no significant relevance.

In the third evaluation step, we read the abstract of each source and apply our in- and exclusion criteria, leaving us with 38 sources relevant for the SLR.

In addition, we ranked the remaining sources according to the ranking criteria “date”, “keyword hit rate in the title”, and “ number of citations”. For each ranking criterion, we have introduced a corresponding metric that assigns a source to a rank: an ordinal scale from “A” (high relevance) to “D” (low relevance). For example, we have assigned the rank “A” to sources with over 1,000 citations. Finally, we ranked all sources based on the mean value of the assigned ranks.

After the ranking, we conducted a full review of the paper for a detailed analysis. We evaluated the quality of the paper and re-evaluated the in- and exclusion criteria, eliminating four additional sources. So in total, 34 (+ seven from the second round) sources made it into the final paper list, which was passed on to the next step.

4.3.3. Extracting the Data

The extraction of data followed the three-step plan defined in the protocol. First, we collected meta-information (e.g., authors and date of publication). Then, we summarised the problems the sources faced by reading the abstract. In the third step, we performed a full review of the filtered sources.

4.3.4. Analysing the Data

After the full paper reviews, we categorised the sources into the pre-defined dimensions (see Section 4.2.2.5). Table 4.2 and 4.3 show the sources and their categories.

Also, we added a column "*Model for*" to the table. Whenever a source targets a modelling approach, the purpose of the model is noted in this column.

Domain	Source	Type	Pattern Type	Language or Technique	Model for
General Software Engineering	[MSM04]	KA	design, programming		
	[ABD+09]	PS			
	[BSA+10]	ER			
	[EE10]	PS			
	[HO09]	SI		ACTORS	
	[HKM08]	PS			
	[ISC+15]	SI		OpenMP	scalability of OpenMP
	[KP11]	SI		auto tuner	
	[MJU+09]	PS	programming		
	[OPT09]	SI		XJava	
	[Pan11]	SI			quality assurance
	[PH11]	SI		auto tuner	
	[PJT09]	ER	design	POSIX, OpenMP	
	[PSJT08]	ER	design	Java, C#, OpenMP	
	[RGD11a]	SI			task, data allocation
	[RGD11b]	ER			
	[SEPA13]	SI			quality prediction
	[SPT10]	SI		auto tuner	
	[SWH+09]	PS			
	[ZP12]	SI		auto tuner	
	[ADKT17]**	SI	design	Framework for stream processing	
	[TMCB16]**	SI	design	optimise data locality	
	[IDSM05]**	SI			Deep Learning Model

PS - Problem Statement
ER - Experience Report
SI - Solution Introduction
KA - Knowledge Accumulation
*Hierarchical State Machine
**Sources from 2020

Table 4.2.: Classification of Sources for General Software Engineering

Domain	Source	Type	Pattern Type	Language or Technique	Model for
HPC	[DMN12]	KA		rage of techniques and languages	
	[HPD09]	SI		OpenMP	
	[Lue08]	PS		CUDA	
	[MGF11]	SI		CUDA, OpenCL	
	[RHJ09]	PS		Shared Memory, OpenMP	
	[CGIP16]**	SI			QN performance model
	[EB16]**	SI			statistic model from empirical obser.
	[PF05]**	SI			Hybrid sim model (DES & MathMod)
	[SEE19]**	SI			QN performance model
Embedded Systems	[BBE+11]	SI		ACTORS	
	[GA12]	PS			
	[LLL+11]	SI	design	VERTAF	
	[GA12]	PS			
	[LCFH14]	SI			HSM*, data parallelism
SPE	[XCDM10]	SI			shared cache
	[THW09]	SI			hierarchical memory
	[VE11]	SI			multiple programs on multi-cores
	[Wil09]	SI			performance counters
	[THW12]	PS			

PS - Problem Statement
ER - Experience Report
SI - Solution Introduction
KA - Knowledge Accumulation
*Hierarchical State Machine
**Sources from 2020

Table 4.3.: Classification of sources for HPC, Embedded Systems, and SPE

4.4. SLR Reporting

In this section, we report on the results of the SLR in detail. Thus, we first give a summary of each paper. The purpose of the abstract is not to fully understand each approach (for this, we refer to the source), but to get an overview of the areas where active research is being done. After the report, we extract valuable lessons learned, summarise the findings, and highlight sources that are particularly relevant as related work for this thesis.

4.4.1. Report Results

Table 4.2 and 4.3 summarises the complete set of sources we found during the search. In the following, we report the results, as we reported them in [FHLB17]. Further, we mark every source that came up while re-performing the SLR with the keyword “**[2020]**”.

The report follows the structure of the domain category.

4.4.1.1. General Software Engineering

"Sources we found for general software engineering address different challenges, which focus mainly on the design and implementation phase in the software development process. For example, in their problem statement, Hwu et al. [HKM08] describe the challenges that arise from concurrent programming and claim that software developers need to apply engineering approaches to handle the complexity involved.

Mehrara et al. [MJU+09] give an overview of parallelism and compiler technology to understand the software development challenge. To ease the development process, the book by Mattson et al. [MSM04] presents a methodical approach for creating parallel programs and gives an overview of patterns. "Finding Concurrency", "Algorithm Structure", "Supporting Structures", and "Implementation Mechanisms" are the four groups of patterns systematised according to the stage of the software development process and reflecting the different abstraction levels during the process. Following this approach, Pankratius et al. [PSJT08] present an experience report on four case studies on developing multicore software for general purpose applications, where each case study uses a different programming language and hardware specification. The report shows that parallelising software is an individual task, and the speed-up can vary. A reason for varying speed-ups is the different hardware specification (i.e., number of cores, cache architecture), which motivates auto-tuners. Auto-tuners are used for source-code-based parallelisation and are addressed by [KP11; PH11; SPT10; ZP12].

Pankratius et al. give another experience report in the form of a case study [PJT09]. Different groups of software developers were asked to parallelise BZip2. Lessons learned are that the use of parallelisation patterns on higher abstraction levels increases the speed-up.

Haller et al. gives another approach [HO09], where a combination of thread-based and event-based models are unified with the help of an abstract ACTOR that provides different kinds of operations to receive messages.

In the work of Iwainsky et al. [ISC+15], the authors automatically generate empirical performance models for OpenMP. They perform tests on different hardware and show that the overhead of OpenMP grows linearly or super-

linearly with the number of threads. Further, they show that the chosen compiler has a major impact on the performance of the application.

To illustrate the hardware impact, Stürmer et al. [SWH+09] compare two different system architectures. In their work, they show that not only the number of cores, but also the memory controller and the caches have a significant impact on performance.

To avoid low-level synchronisation defects during the software development, new programming languages are proposed. For example, XJava, which preserves the object-oriented approach while simplifying the expression of parallelism, is presented by [OPT09]. To support the development of new programming languages, an automated usability evaluation for the design of parallel programming languages was introduced by Pankratius [Pan11].

Rodrigues et al. [RGD11a] utilise a meta-model extension on MARTE profiles to specify the task and data allocation in the memory hierarchy for GPU architectures.

We also found an experience report by Rodrigues et al. [RGD11b] that describes a case study where the authors use UML and the MARTE profile to specify and generate OpenCL code with the help of model-driven engineering approaches. They claim that the model-driven engineering approach is well suited for programmers to create parallel programs and that the MARTE profile has a high potential for parallel modelling programs.

The paedagogically-oriented contribution of Brown et al. [BSA+10] focuses on the education of 'new generations of students'. They identify a list of recommendations to improve students' knowledge of parallel programming.

The work of Sagardui et al. [SEPA13] needs to be highlighted because it focuses on verification and validation of multicore systems in early design phases. In their related work, they show that in the embedded system domain, there are approaches for modelling multicore systems with the help of MARTE profiles. Their contribution is a high-level process, which recommends the use of three models to represent multicore systems and their software: an application model, a platform model, and an allocation model."[FHLB17]

[2020] An additional three sources came up in this category when re-performing the SLR in 2020: In his doctoral dissertation [TMCB16], C. Ter-

boven describes the high relevance of data locality to the performance of parallel applications. He develops an approach to optimise the data locality in NUMA systems for the OpenMP paradigm. For this purpose, he creates a thread-affinity model. In [ADKT17], the authors face the issue of the absence of 'good high-level programming tools'. To overcome this problem, they introduce FastFlow, which is a framework that uses a steam-based paradigm to parallelise. They enable the software architect to model their system using cyclic graphs. Finally, [IDSM05] proposes a deep learning approach to estimate the performance of a parallel application by using multilayer neural networks.

4.4.1.2. HPC

"All sources we found in the HPC domain focus on techniques to enable parallelism in HPC applications. Diaz et al. [DMN12] performed a survey. They comprehensively described different concepts, libraries, and languages to bring parallelism to applications. They show that distributed memory is the most commonly used programming approach for parallel programming in the HPC domain. Further, the work from Rabenseifner et al. [RHJ09] focuses on the potentials and challenges of this dominant programming model.

Other sources we found introduce problem-specific solutions to handle parallelism. Hadjidoukas et al. [HPD09] introduce a user-level thread library called PSTHREADS, which allows the use of fine-grained parallelism with large numbers of threads. Luebke et al. [Lue08] explained the CUDA programming model and argued for its use in the biomedical imaging community. Martinez et al. [MGF11] proposes a source-to-source translator from CUDA to OpenCL."[FHLB17]

[2020] During the re-performance, we found four additional sources, all highly relevant. Two of them [CGIP16; SEE19] use a Queuing Network (QN)-based approach. More specifically, [CGIP16] uses QN along with both analytical and simulation-based solvers to optimise parameters for parallel execution in systems with CPUs and GPUs. In contrast, [SEE19] uses QN along with non-linear solvers to estimate the message communication for Message Passing Interface (MPI)-based applications in cloud environments.

They focus on the interaction delay in distributed systems caused by the network delay.

In [EB16], the authors use a statistical approach to estimate the performance of parallel applications. They perform small-scale experiments, measure and analyse these small-scale experiments, and use these data to estimate the performance in large-scale scenarios. Finally, the authors of [PF05] combine discrete event simulations and mathematical modelling to create a performance model for parallel and distributed systems. Further, they use UML activity diagrams to model the low-level (close to code) behaviour of the application and enrich it with additional performance relevant information.

4.4.1.3. Embedded Systems

"The majority of the sources we found in the domain of embedded systems introduce an approach to handle parallelism within a program. Bini et al. [BBE+11] present the approach developed in the ACTORS project. They show that the ACTORS approach is useful in handling time-sensitive applications with variable load. A problem statement paper by Gray et al. [GA12] describes the challenges of multicores in the embedded domain on the model-driven software engineering level. They identify problems within the whole development spectrum (i.e., system modelling, programming models of software languages, analysis and verification, toolchains support, and sophisticated hardware implementations).

Llopard et al. introduce a modelling approach. [LCFH14] that combines hierarchical state machines (HSMs) with data parallelism and operations on compound data.

Lin et al. [LLL+11] propose a framework to generate program code for multicore embedded systems out of SysML models."[FHLB17]

4.4.1.4. SPE

"After the final evaluation, five sources in the SPE domain remained. As one would expect, all the sources focus on improving the accuracy of performance prediction for multicore systems by either adopting an existing performance model or proposing a new model.

In [THW09], Treibig et al. improve the predictive power by including properties of cache hierarchy design with the use of the simple balance metric. Further, the authors publish a problem statement paper [THW12] and discuss the sensible use of hardware performance counters in a structured performance engineering approach. Additionally, typical performance patterns and their respective metric signatures are defined.

Xu et al. [XCDM10] propose a new performance model called CAMP for shared memory on multicore systems. The model uses non-linear equilibrium equations.

Van Craeynest et al. [VE11] also proposes a new model, MPPM, for estimating multi-program multicore performance. It employs a method to model the performance entanglement between co-executing programs with shared caches.

Samuel Williams uses a roofline function to determine the correlation between floating-point operations and bytes transferred from DRAM to estimate the peak performance of a CPU in [Wil09]." [FHLB17]

4.4.2. Evaluate Report

In the previous section, we presented insights into our search results. To sum up our findings, we derive the critical lessons learned during the SLR [FKB18]:

Programming Languages: Especially in the software engineering domain, we found various approaches that introduce new programming languages [HO09; OPT09; Pan11], which are supposed to ease the development process and raise the level of parallelism from a low, code-based level, to a design level.

Patterns: In the software engineering discipline many patterns exist to tackle parallelism on different abstraction levels [MJU+09; MSM04; PJT09; PSJT08]. Programming patterns are useful to help software developers implement software in a faster and more structured way. Design patterns help software developers bring parallelism to multiple levels of software design. Both types of patterns help software developers to abstract the degree of parallelism.

Libraries: Libraries are common in the HPC domain to address large-scale parallelism [HPD09; Lue08; MGF11; RHJ09]. Most common are distributed memory approaches like MPI, but approaches like CUDA are gaining importance [DMN12]. In recent years, these libraries also have become more critical in the embedded system domain and general software development.

Auto-parallelisation: In addition to parallel programming, much research has been conducted in auto-parallelisation on a low abstraction level (e.g., compiler).

Auto-tuner: Auto-tuners optimize software for various hardware/ architectures and are still under heavy development [KP11; PH11; SPT10; ZP12].

UML and MARTE Profiles: Research is being performed in the field of verification and validation of multicore enabled software. Further, several approaches exist to model multicore systems with the help of UML and MARTE profiles, but to date none of these approaches supports performance prediction in an SPE way [RGD11b].

Technical Focus of HPC: All sources we found from the HPC domain focus on a close-to-programming level. This observation leads to the hypothesis that high-level modelling is not conventional in the HPC domain. Our expertise supports this hypothesis. Another reason for this result could be the selection of search terms (see also Section 4.5).

Performance Prediction: The approaches and models we found for performance prediction mostly focus on adopting models to include shared memory [THW09; VE11; XCDM10] or memory bandwidth behaviour [Wil09] to increase the prediction accuracy. However, the majority of the sources claim only to provide initial work.

HPC Modelling: When re-performing the SLR we found four approaches [CGIP16; EB16; PF05; SEE19] in the domain of HPC using models to evaluate quality attributes of parallel and distributed software. In contrast to that, we did not find any model-based approach in the first run. That indicates an increased awareness and need for performance models in that domain.

In addition to the above insights, we can answer the SLR's research questions as follows:

R$_{SLR-1}$**:** We found various modelling approaches that try to consider parallelism on a model level. For this, the authors create their modelling languages or extend existing ones (like UML) with constructs like the UML Profile for MARTE. Overall, only one source [PF05]—from the HPC domain—utilises the model adoptions for performance prediction.

R$_{SLR-2}$**:** We found five sources proposing approaches to predict the performance of multicore systems. Three of these approaches include memory designs to their models. One method uses a roofline function to determine the correlation between floating-point operations and bytes transferred from DRAM, and one uses hybrid solvers to simulate the performance of low-level algorithmic problems.

4.5. Threats to Validity

During the design of the SLR, we made several decisions according to our scope. Each one brings certain trade-offs, which we discuss in the following, and as was reported in [FHLB17]:

Search Terms: In Section 4.2.2, we describe how we derived the search terms. For each search term, we created a synonym list. The list was discussed with experts from at least two domains. Based on the fact that our search covered even more domains and that synonyms are commonly used, we cannot guarantee that we included all possible combinations.

Search Engine: In Section 4.2.2, we also decided to use Google Scholar as a search engine because Google Scholar works as a meta-search engine that covers a wide range of databases. To minimise the risk, we performed test searches with other search engines like SpringerLink, ACM Digital Library, or IEEExplore. The results indicate that Google Scholar covers them as well. However, using other or additional search engines might bring different or additional results.

Pre-selection : Due to the problem that strictly applying the blacklist to the search results led to very few results (see Section 4.3.2), we decided to use manual pre-selection. Even when the pre-selection was based on personal experience, we assume that the exclusions are mostly correct. Based on the fact that the number of papers we kept is much higher than that yielded by automatic pre-selection, we believe that we attained a higher accuracy.

Date Restrictions: In the review protocol (Section 4.2.2), we limited the sources considered to those published between 2003 and 2020 because we wanted to focus on developments after multicores came into common use in desktop computers. Considering sources before 2003 might bring additional results.

Re-performing: When re-performing the search in 2020, we followed the initial review protocol strictly, to ensure comparable results. To only capture the delta of sources, we focused on sources published from 2016 onward. However, we noticed two sources from 2005 that had not shown up in the first search. We decided to include these sources as well, even though we cannot explain why they did not show up in the first search. One reason might be copy right related reason which run out by now.

4.6. Summary

The SLR revealed useful insights in the area of parallel programming, parallel modelling, and parallel performance prediction. Even though none of the approaches satisfy our requirements, we gained a lot of insights and knowledge in this area. On top of that, we acknowledge the work in three areas, especially:

Parallel Modelling: It becomes clear that the expression of parallel behaviour in software models is more and more relevant. Therefore, [RGD11b] aims to use UML MARTE profiles to enrich software models with multicore information. Even though they do not focus on performance predictions, but on code generation for OpenCL, and therefore focus

on a low abstraction viewpoint, their ideas and methods to express parallel behaviour might be adaptable.

Even more relevant is the work from [PF05], in which they use UML activity diagrams to specify software performance models for parallel applications. Again, they focus on low abstraction levels and assume an implementation already exists, but these insights should be used to create performance prediction models on architectural levels during the design phase.

Analytical Performance Models: Numerous approaches exist to use either analytical models [THW09; VE11; XCDM10], statistical models [EB16; Wil09], or QN [CGIP16; SEE19]. All of these approaches have in common that they focus on cache, bandwidth, or memory interaction, particularly on simplified and low-level scenarios.

Performance Prophet: In the work of Pllana et al. [PF02; PF05], the authors introduce a novel approach to use a hybrid variant of analytical and simulative performance models. They use UML activity diagrams to model the behaviour of procedural modelling languages (e.g., C or Fortran). Additionally, they use cost functions to specify the resource demands and hardware capabilities. Their main goal is to predict the performance of MPI-based scientific applications on large-scale multi-node hardware environments. Noteworthy is the differentiation between node internal and external behaviours, which are handled by either event-driven simulators or analytical solvers. The major drawback is the estimation of the cost function. The cost functions are an essential part of the performance model. However, their estimation is far from trivial. We will address this topic in CB_1 and CB_4 as well. Further, we take the insights from [PF02] into account when proposing additional language constructs for parallel software behaviour in performance engineering.

In addition to the references revealed by the SLR, there exist other related works that are not directly related to performance predictions of parallel applications. The exacted schedulers from J. Happe [Hap08] and the CloudSim project are the two most relevant contributions [CRB+11].

J. Happe included a concept of exacted scheduling in the performance prediction approach. This approach takes several effects (like overhead for content

switches) for specific scheduling approaches into account. The approach is designed to work mainly for single cores and does not addresses the challenges of multicore systems. However, it supports concurrent software, and can therefore be used for parallel applications as well. Even though this increases the prediction accuracy of parallel applications, the impact is rather small [FH16].

Like Palladio, the CloudSim approach is a system simulator for cloud environments. Similar to Palladio, CloudSim uses a specification of a hardware, software and usage model to simulate quality attributes like response time and elasticity of cloud environments. Due to this characteristic, they both support basic parallel executions of containers. However, they do not consider multicore aspects and assume a linear speedup.

Since none of the related work satisfies our requirements or answers our research question, we take the insights from the SLR into account and continue with our research process (see Figure 3.2).

5. Running Example: Resource Demands

Through the course of the thesis, we refer to different kinds of representative examples. In this section, we introduce each example, give a brief description, an implementation example, and characterise it. We categorise the examples into two groups: Resource Demanding Examples and Complex Examples.

5.1. Resource Demanding Examples

The group of resource-demanding examples represents very low-level and algorithmic examples, where each represents a special kind of resource-demanding behaviour. Most of the resource demands can be marked as processor-intensive demands (which mainly consume CPU time), I/O intensive tasks (which have many reads and writes, memory accesses, and consume memory bandwidth), or a characteristic combination of both. We will not focus on an optimised implementation of the given problems for a specific hardware. Moreover, we are interested in the characteristics of resource-demanding behaviours, since this will be relevant for the performance predictions later. In the following, we will briefly explain each resource demand and give an implementation example. Further, the realisation of each resource demand in Protocom is provided in Appendix A.2. The general implementation example will help to understand the core problem. In contrast, the implementation from Protcom will help in following Contribution 4 (see Chapter 7).

5.1.1. Fibonacci Numbers

Description: In mathematics, the Fibonacci numbers (or Fibonacci sequence) is a well-known sequence and describes the addition of two preceding numerical values to get the current value. The first element of the sequence is $F_0 = 0$ and the second i $F_1 = 1$. For all other elements $F_n = F_{n-1} + F_{n-2}$ must hold [Knu97].

Implementation: Implementing a sequential version of the Fibonacci sequence is straightforward, and a Java implementation is given in Lst. 5.1. In this implementation, recursion is used to calculate all the other numbers up to the given position. This implementation shows the core problem and is not optimised for the most performant execution.

```
1   /* Returns the fibonacci number of the position n
2    * in the sequence. */
3   static int fibonacci(int position) {
4     if (position <= 1) {
5       return position;
6     }else {
7        return fibonacci(position-1) + fibonacci(position-2);
8     }
9   }
```

Listing 5.1: Sample implementation of the Fibonacci number in Java

Since the Fibonacci number of position n is based on the two preceding numbers, a parallelisation of this problem is complex and exceeds the scope of this work.

Characterisation: The actual work of the Fibonacci number calculation is a simple addition. Therefore, the Fibonacci demand is a processor-intensive demand [FBKK19]. Storing the preceding values is very low overhead and can be done efficiently in L1.

5.1.2. Mandelbrot Set

Description: The Mandelbrot Set is another mathematical sequence, which is named after the French mathematician Benoit Mandelbrot (cf. [DH84]).

The sequence is a set of complex numbers which is defined by the iteration of $z_0 = 0$ and $z_n + 1 = Z_n^2 + c$.

Geometrically interpreted as a part of the Gaussian number plane, the Mandelbrot set is a fractal. Images of it can be generated by placing a pixel grid on the number plane and assigning a value of c to each pixel. If the sequence is restricted with the corresponding c, i.e. if it belongs to the Mandelbrot set, the pixel will be coloured (e.g., black), and otherwise not. If the colour is determined by how many elements of the sequence have to be calculated until it is clear that the sequence is not restricted, a so-called speed picture of the Mandelbrot set is created: The colour of each pixel indicates how fast the sequence with the respective c is heading towards infinity.

Implementation: The following implementation (Lst. 5.2) plots a region (size by size) of the Mandelbrot set. The variables xc and yc represent the centre of the region, while n gives the size dimension and max defines the maximum number of iterations.

```
1  public class Mandelbrot {
2
3     // return number of iterations to check if c = a + ib is in Mandelbrot set
4     public static int mand(Complex z0, int max) {
5        Complex z = z0;
6        for (int t = 0; t < max; t++) {
7           if (z.abs() > 2.0) return t;
8           z = z.times(z).plus(z0);
9        }
10       return max;
11    }
12
13    public static void main(String[] args) {
14       double xc = Double.parseDouble(args[0]);
15       double yc = Double.parseDouble(args[1]);
16       double size = Double.parseDouble(args[2]);
17
18       int n = 512; // create n-by-n image
19       int max = 255; // maximum number of iterations
20
21       Picture picture = new Picture(n, n);
22       for (int i = 0; i < n; i++) {
23          for (int j = 0; j < n; j++) {
24             double x0 = xc - size/2 + size*i/n;
25             double y0 = yc - size/2 + size*j/n;
26             Complex z0 = new Complex(x0, y0);
27             int gray = max - mand(z0, max);
28             Color color = new Color(gray, gray, gray);
29             picture.set(i, n-1-j, color);
30       } }
```

```
31            picture.show();
32    } }
```

Listing 5.2: Sample implementation of the Mandelbrot Set in Java [SW17]

Characterisation: Creating a graphical representation of the Mandelbrot set is not only a computation-intensive task, a lot of (complex) numbers have to be calculated, stored, and re-accessed as well. Therefore, the Mandelbrot Set demand can be characterised as I/O-intensive task.

5.1.3. Sorting Arrays

The sorting array demand is characterised by a lot of data access and swap-or-switch operations. In practice, a lot of different sorting algorithms are known, and have different pros and cons. In the following, we will focus on the Dual Pivot Quicksort algorithm, since this one is also implemented in the Java base class library.

Description: The Dual Pivot Quicksort algorithm [Yar09] is an improved version of Quicksort. It is characterised by using two pivot elements, one at the left end of the array and one at the right end of the array. In this algorithm, the left element must be smaller or equal to the right element. Otherwise, they will be swapped. After that, the set is spilt into three subsets: Values smaller than the left pivot element, values larger than the right pivot element, and values between the left and right element. After that, the three sets are partitioned and step one is repeated until all partitions contain only one element. At the last step, they are merged.

Implementation: The following code in Lst. 5.3 is an implementation of the algorithm described above from the Java base class library. The code is highly optimised and hard to read. An easily comprehensible version, along with detailed explanations, can be found in [Yar09].

```
1   static void sort(int[] a, int left, int right,
2                    int[] work, int workBase, int workLen) {
3          // Use Quicksort on small arrays
4          if (right - left < QUICKSORT_THRESHOLD) {
5             sort(a, left, right, true);
6             return;
7          }
8
9          /*
10          * Index run[i] is the start of i-th run
11          * (ascending or descending sequence).
12          */
13         int[] run = new int[MAX_RUN_COUNT + 1];
14         int count = 0; run[0] = left;
15
16         // Check if the array is nearly sorted
17         for (int k = left; k < right; run[count] = k) {
18            if (a[k] < a[k + 1]) { // ascending
19               while (++k <= right && a[k - 1] <= a[k]);
20            } else if (a[k] > a[k + 1]) { // descending
21               while (++k <= right && a[k - 1] >= a[k]);
22               for (int lo = run[count] - 1, hi = k; ++lo < --hi; ) {
23                  int t = a[lo]; a[lo] = a[hi]; a[hi] = t;
24               }
25            } else { // equal
26               for (int m = MAX_RUN_LENGTH; ++k <= right && a[k - 1] == a[k]; ) {
27                  if (--m == 0) {
28                     sort(a, left, right, true);
29                     return;
30                  }
31               }
32            }
33
34            /*
35             * The array is not highly structured,
36             * use Quicksort instead of merge sort.
37             */
38            if (++count == MAX_RUN_COUNT) {
39               sort(a, left, right, true);
40               return;
41            }
42         }
```

Listing 5.3: Implementation of the sort method of the DualPivotQuicks from the Java base class library

Characterisation: Due to the high interaction with the memory and the enormous amounts of reading and writing operations, the Dual Pivot Quicksort algorithm is a highly I/O-intensive task.

5.1.4. Calculating Primes

Description: In mathematics, a prime number is a natural number, that is higher than one, and that cannot be formed by multiplying two smaller natural numbers.

Prime numbers are of high interest in informatics, especially in cryptography. Large prime numbers are used for encryption.

Even though there are different approaches to find a prime number, i.e., trial division (i.e., brute force) or with the help of the Sieve of Eratosthenes [One09], it remains a resource-intensive task. The current largest prime number is $2^{82,589,933} - 1$ and was discovered by Patrick Laroche in 2018 [Lar18].

Implementation: The implementation in Lst. 5.4 shows a trial division approach to find prime numbers. It simply checks whether each number is divisible by another number higher than one.

```
1   public static List<Integer> getPrimeNumbers(final int upperBound) {
2      List<Integer> resultSet = new ArrayList<>();
3      for (int i = 2; i <= upperBound; i++) {
4         if (isPrime(i)) {
5            resultSet.add(i);
6         }
7      }
8      return primeNumbers;
9   }
10  public static boolean isPrime(final int numberToCheck) {
11        boolean result = true;
12     for (int i = 2; i < numberToCheck; i++) {
13        if (numberToCheck % i == 0) {
14           result = false;
15        }
16     }
17     return result;
18  }
```

Listing 5.4: Implementation of trial division approach to calculating primes

Characterisation: The base characterisation of the above calculating prime resource demand is defined by the method `isPrime`, which performs a high number of divisions. This leads to a load on the CPU. The I/O interaction

is comparably low. The few numbers can even be stored in caches. Thus, calculating prime demand is a CPU-intensive demand.

5.1.5. Counting Numbers

Description: Counting numbers is a straightforward algorithm to count numbers from zero upwards toward a limit. This example is a synthetic demand, which is added here because it can put much pressure on the memory architecture.

Implementation: The implementation of the counting number example is given in Lst. 5.5. It shows a for-loop which iterates until the given upper limit is reached. In each iteration, the current counter i is added to a counting variable of k. In this Java implementation, k must be a class variable to prevent the just-in-time compiler from removing it during the code execution—as part of the just-in-time code optimisation.

```java
1   // needed to stop the JIT compiler from removing the code in execute
2   private long k;
3
4   private void countNumbers(final double countTo) {
5      for (long j = 0; j < countTo; j++) {
6         if (k > 100000) {
7            k = 0;
8         }
9         k += j;
10     }
11  }
```

Listing 5.5: Implementation of the counting numbers demand from Protocom

Characterisation: The characteristics of the counting number demand are rather simple but at the same time interesting. The demand produces both CPU demand from the addition and I/O demand by getting the numbers from memory. The latter can be neglected when executing the code sequentially because the numbers will be stored in L1 or registers.

5.1.6. Matrix Multiplication

Description: In mathematics, matrix multiplication is a multiplicative combination of matrices. To multiply two matrices with each other, the number of columns in the first matrix must match the number of rows in the second matrix. The result of matrix multiplication is again a matrix. The entries of the new matrix are calculated by multiplying and summing the entries of the rows of the first matrix, component by component, with the columns of the second matrix.

Matrix multiplication is often used in linear algebra or natural science. Each c_{ik} entry of the matrix product is calculated by $c_{ik} = \sum_{j=1}^{m} a_{ij} \cdot b_{jk}$. In this equation, a_{ij} and b_{jk} are the corresponding entries of the matrices A and B, when AxB is calculated.

Implementation: The implementation in Lst. 5.6 shows an example of a matrix multiplication. The number of columns of matrix a must be equal to the number of rows of matrix b.

```
1  public static int[][] multiplyMatrix(final int[][] matrixA,
2                                       final int[][] matrixB) {
3     int[][] result = new int[matrixA.lenght][matrixB[0].length];
4     for (int i = 0; i < matrixA.length; i++) {
5        for (int j = 0; j < matrixB[0].length; j++) {
6           for (int k = 0; k < matrixA[0].length; k++) {
7              result[i][j] = result[i][j] + matrixA[i][k] * matrixB[k][j];
8           }
9        }
10    }
11    return result;
12 }
```

Listing 5.6: Example implementation of the a matrix multiplication in Java

Characterisation: Matrix multiplication is a good example of an I/O intensive task because for each multiplication, two values have to be loaded from memory, and one value has to be written. The multiplication itself has only a moderate impact on the CPU. Further, the order of the three for-loops has a significant impact on performance. Arranging the i,j,k properly can cause caching effects because the data of arrays are stored in the main memory in a way that the next value is within the same cache page and proactively

loaded (see page cache for more details). Arranging them in the wrong order will result in a lot of cache misses and main memory access, which results in a degraded performance. The difference between the best and worst version can impact the performance by a factor of eight (the worst combination is eight times slower than the best combination) [FH16].

5.1.7. Summary

The examples given in this section will be used throughout the further course of this thesis, each example representing a unique resource demand. Table 5.1 summarises the characteristics of the individual demands and gives the CPU-intensity and I/O intensity of each demand.

Resource Demand	CPU-intensity	I/O-intensity
`FibonacciNumbers`	high	low
`MandelbrotSet`	low	high
`SortingArrays`	low	high
`CalculatingPrimes`	high	low
`CountingNumbers`	low	medium
`MatrixMultiplication`	medium	high

Table 5.1.: Summary of Resource Demand Characteristics

5.2. Complex Examples

In the upcoming section, we will describe more complex examples which produce a more extensive resource demand. The first example—Bank Transaction—is a common one, when it comes to interaction between multiple threads or actors.

The second example is taken from the SPEC Benchmark Suite. It consists of multiple combined low-level demands (like the ones explained before). SPEC Benchmarks are often used to evaluate the performance of hardware systems. Thus, they can be used as a substitution for more complex real-world examples. The advantage of using SPEC Benchmarks instead of real

examples is that they are (a) more comfortable to set up, and (b) better compared to different setups.

5.2.1. Bank Transaction Example

Description: The bank transaction example is a common example used in literature to describe various problems in parallel execution [Lin10a]. Its underlying data model consists of a simplified version of the bank domain. Figure 5.1 shows the domain represented by a UML class diagram.

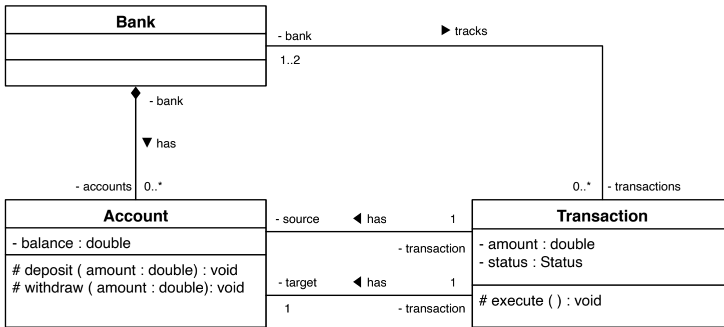


Figure 5.1.: Domain View of the Bank Transaction Example (cf. [Lin10a])

In this example, a bank consists of a set of accounts. Each account has a balance and a method to deposit or withdraw money. Further, there are transactions which transfer a specific amount of money from one account to another. A transaction is successful if the balance of the source account is higher than the amount of the transaction and the money can be transferred. Vice versa, a transaction will fail if the balance is insufficient.

The scenarios rising from the example are complex—especially for parallel executions—because the order in which the transactions are executed is important. Additionally, it must be guaranteed that only one transaction is executed for a bank account to prevent multiple write operations simultaneously.

Implementation: For this example, a variety of instances can be found across the literature. However, in the course of this thesis, we will refer to the version conceived by J. Link [Lin10b]. Link uses AKKA Actors to implement the scenario. As introduced in Chapter 2, Actors are used as a means to parallelise. In the example presented, each bank account represents an actor with its own message queue. In the message queues, the incoming transactions are stored. Further, Link uses a transaction actor, which manages the individual transactions. Thereby, each transaction is executed in the following order: (1) get the source and target account, (2) check account balance, (3) withdraw money and (4) deposit money. Given the use of the actor paradigm, the example implementation can be executed in parallel, and multiple transactions are processed at once.

The full implementation can be found in [Lin10b].

Characterisation: The primary work in this example is subtraction or addition. However, the use of Actors puts much additional overhead on top. Every message utilises the memory bus and uses additional memory. So, if we consider the example by using an Actor implementation, we can expect a low to medium demand on the CPU and a comparable high demand on the memory architecture.

5.2.2. SPEC Benchmarks

In two seminar theses we evaluated in collaboration with P. Gruber [Gru20] and A. Yoon [Yoo19], the suitability of performance benchmarks as use case examples. The following sections are part of these works:

Performance benchmarks (e.g., from SPEC[1]) are designed to evaluate the performance of computer systems. Further, they can be used to make different computer systems comparable. To ensure comparability, a benchmark is standardised and portable, which means the benchmark has the minimum possible dependencies on specific hardware. Additionally, benchmarks are not designed to stress the operating system. Depending on the benchmark set, it stresses the graphic card, the I/O bus, or—most commonly—the CPU.

[1] `https://www.spec.org/`

According to SPEC (the Standard Performance Evaluation Corporation)—a non-profit organisation whose goal it is to establish, maintain and endorse a standardised set of relevant benchmarks for computer systems—a benchmark is "a standard of measurement or evaluation" [SPE20]. A computer benchmark refers to a computer program which executes a set of operations to produce a metric that represents the performance of a computer environment. A Benchmark typically measures execution speed and throughput as metrics. These metrics are used to analyse the performance of a system [SPE20].

Running the same benchmark on different hardware enables us to compare the performance of the different systems [SPE20]. According to the IBM Knowledge Centre, benchmark testing can help to determine current performance (issues) and help to improve performance [IBM18].

In the following, we will focus on the SPEC Benchmark sets, as they are very commonly used. However, other benchmark sets are suitable as well.

When the user runs the benchmarks from SPEC, he usually gets a base and a peak value for the specific task. The main difference between base and peak is that peak is the result of using optimisations for the particular task, while the base value is based on the same optimisation setting for all tasks [MVL+10]. In general, both are reflecting the time the task has run. Further, the benchmark outputs the ratio between the execution time and the run time of the benchmark on a reference system. The creators of the benchmark individually chose the reference system. This ratio would give an impression of whether the used system were faster, slower, or as fast as the reference system. This allows the evaluation of comparison results at first glance. In the end, the SPEC benchmarks deliver a specification which ideally gives an impression of how well a system performs. The general specification is the median value of all applications.

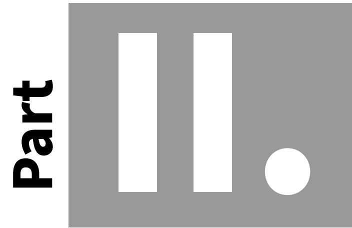
In the following, we will give a brief overview of the SPEC benchmarks, focusing on parallel execution as they are suited to test multicore systems.

SPEC MPI 2007: SPEC MPI consists of 13 different applications [MVL+10]. All 13 applications are examples from a scientific background. They are used to perform weather predictions or to simulate fluids. They are all implemented in FORTRAN or C(++). In contrast to the above resource-demanding examples, these tasks are neither low, in terms

of complexity, nor created synthetically. The SPEC MPI benchmark uses MPI calls as a means to parallelise. That means the independent processor cores need to communicate with each other regularly. Müller et al. [MVL+10] give additional details about the message size, implementation, and number of message calls.

SPEC OMP 2012: SPEC OMP is a benchmark built upon the OpenMP framework and uses shared memory instead of message passing. It consists of a total of 15 applications and also includes optional power consumption metrics—in addition to the base and peak metrics [MBB+12].

SPEC ACCEL: This benchmark consists of 49 applications in total, and uses different approaches to parallelise. So, 19 applications use OpenCL, 15 applications use OpenACC, and 15 applications use OpenMP. In comparison to the above benchmarks, this benchmark set does not focus on the CPU and CPU architecture but on the GPU—which is not in the scope of this thesis.



Contributions

6. CB_1: Parallel Architectural Pattern Catalogue

In the previous sections we learned about the foundations and state of the art of parallel computing, hardware architectures, and parallelisation paradigms, defined the research approach and the research question to be answered in this thesis, and followed the research design. In the next four chapters we lay out the individual contributions (numbered from CB_1 to CB_4 according to the RQ_1 to RQ_4) in detail.

The first contribution picks up the requirement $R_{modelling}$ deployed in Chapter 1. The requirement is that software architects should be able to express concurrency in software models in a way that characterises the behaviour of the software. The specification also includes highly concurrent software with multiple thousands of concurrently executed threads. As a result of this chapter, we can present an answer to the research question RQ_1, validate hypothesis H_1, and present a parallel architectural pattern catalogue, which contains reusable knowledge. Given that pattern catalogue, the software architect can easily and efficiently model the behaviour of parallel software.

Figure 6.1 lays out the process followed to produce the first contribution. As the first step of this process, we analyse the current state of the art and establish why this requirement is currently not fulfilled. Next, we define a set of challenges to overcome and goals to meet in order to fulfil the requirement. To evaluate the quality of a parallel modelling language enhancement, we propose a set of evaluation metrics next. After that, we investigate different strategies to enhance current modelling languages, pick the most suitable one for our scenario, and execute the approach using the example of OpenMP parallel loops.

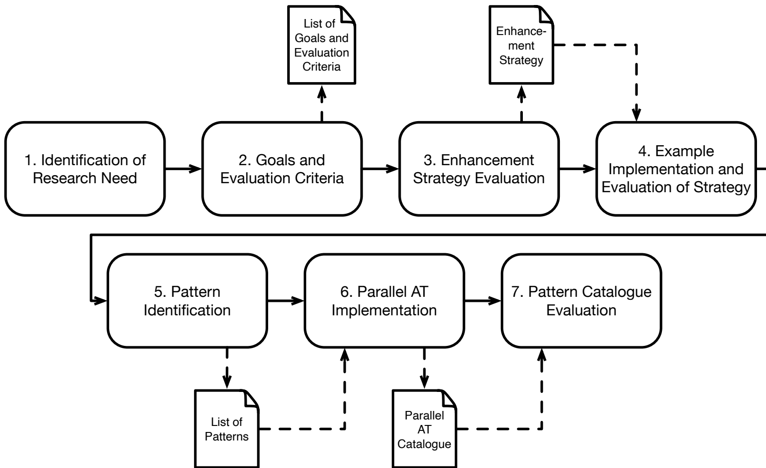


Figure 6.1.: Overview of the Research Method for Contribution C_1

After confirming that the strategy is suitable, we identify a list of the most useful parallel patterns, which we implement and combine in the parallel architectural template catalogue. Finally, we execute an empirical study to evaluate this catalogue.

As result we present a parallel architectural pattern catalogue containing three of the most frequently-occurring parallelisation patterns. We can show that the use of the pattern catalogue increases the efficiency of the SAs significantly. Furthermore, we are able to increase the accuracy of the performance predictions with the help of overhead functions.

Please note that significant parts of the work from step 1 are reviewed and published in [FH16]. Additionally, the results from steps two through four are summarised, published, and reviewed in [FKHB19]. Finally, the specification of the pattern behaviour (described in Section 6.6) is reviewed and published in [FHB20].

All raw data, implementations, and accompanying resources are publicly available:

Section 6.1 Peformance Prediction for Matrix Multiplications:
`https://zenodo.org/badge/latestdoi/250200347`

Section 6.5 Parallel Architectureal Pattern Catalogue:
`https://github.com/PalladioSimulator/Palladio-Addons-ParallelPerformanceCatalogue`

Section 6.7.2 User Study Data:
`https://doi.org/10.5281/zenodo.3755339`

6.1. Problem Space

To emphasise the issues with current modelling approaches, we will first briefly report on a controlled experiment we performed in [FH16][1]. In this work we used a matrix multiplication example (see Section 5.1.6). Later we will leverage the same example to evaluate our enhancements to existing modelling languages.

6.1.1. General Information

In the controlled experiment, we evaluate the multicore and multi-threading capabilities of the current state-of-the-art performance modelling tools. In this specific case, we prioritise Palladio and raise the following research questions:

RQ_{p1} Is it possible to model multicore systems with Palladio?

RQ_{p2} How precise are the predictions?

[1]The full experiment description can be found in [FH16], and all data are available at `https://zenodo.org/badge/latestdoi/250200347`

To answer these questions, we confront the problem from two sides. On the one side, we implement a matrix multiplication as a parallelised code example and measure the execution time (response time) on dedicated hardware. On the other side, we model the same instance with Palladio and perform a simulation.

As a metric, we focus only on the execution (or response) time of the actual matrix multiplication. To evaluate its accuracy, we compare measurements to our simulation result.

6.1.2. Implementation

Listing 6.1 shows the implementation we used for the matrix multiplication. The implementation follows the explanation in Section 5.1.6 and uses three for-loops to multiply and add up the respective matrix elements of `matrixA` and `matrixB`. The provisional sum is stored in `matrixC`. When all iterations are finished, `matrixC` holds the results of the matrix multiplication. The order of the three for-loops can be altered without changing the result, but this impacts the performance greatly. We tested all variants on our target hardware and chose the fastest variant as described in [FH16].

For parallelisation, we used a framework—the omp4j[2] framework. It provides basic OpenMP functionalities like parallel sections and loops for the Java environment, and supports up to 16 worker threads. To use this framework we simply had to add `line 5` to the code and use the omp4j pre-compiler (see Lst. 6.1). Please note that the `threadNum` feeds omp4j the number of threads it should use. This parameter is optional. We used that number to set the number of threads. When not specified, the default is the number of available CPU cores. The scheduling parameter is optional. A static scheduling tells omp4j to do the scheduling while pre-compiling and not during runtime (dynamic).

```
1  /* Requires: matrixA, matrixB, matrixC != null;
2   * Requires: matrixA.getWidth == matrixB.getHeight;
3   * Ensures: matrixC = matrixA x matrixB;
4   */
5  // omp parallel for schedule(static) threadNum(2)
6  for (int i = 0; i < matrixA.getWidth(); i++) {
7    for (int k = 0; k < matrixB.getHeight(); k++) {
```

[2]See `http://omp4j.org` and `https://github.com/omp4j/omp4j`

```
8       for (int j = 0; j < matrixA.getHeight(); j++) {
9         result[i][j] += matrixA[i][k] * matrixB[k][j];
10  } } }
```

Listing 6.1: Sample implementation of a matrix multiplication in Java with OpenMP annotations

6.1.3. Modelling

While implementing the matrix multiplication is straightforward, the modelling part is more challenging. To model the software behaviour in Palladio, we need to know some characteristics of our software; for example, the resource demand (i.e., the CPU time) for a specific task like a single multiplication, and how often this action is performed. Tasks that demand a single resource are called actions in Palladio.

To gather the additional characteristics, we first measure a sequential matrix multiplication and estimate the resource demand for a single multiply-add (line 9). We compute the number of multiply-add operations from the input matrices dimensions.

With this information at hand, we begin to model the use case, starting with a sequential version. Figure 6.2 shows the PCM's Service Effect Specification (SEFF), which we use to model the software behaviour. The SEFF consists of only one action, which includes the resource demand for one multiplication (`0.00000069`) multiplied by the number of multiplication operations needed (indicated by the input matrices' dimensions). We took the resource demand from the measurements and it represents the time it takes to perform a single multiply-add operation.

We could also have used three nested PCM loop-actions and only annotated the actual resource demand in the internal action, which would be a more natural approach. However, we chose the first approach because it abstracts the actual algorithm and greatly improves performance during analysis [FH16].

After creating the sequential model, we adapt it to fit the parallel scenario. This process involves much manual modelling, since the parallel constructs in the PCM are aligned to UML and are therefore very basic (e.g., do not

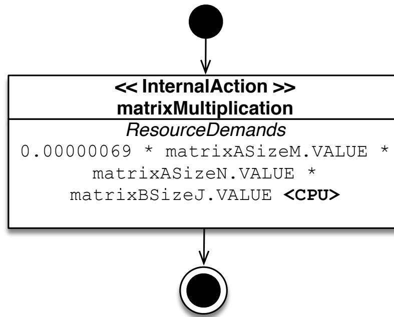


Figure 6.2.: SEFF Definition of the Sequential Model

support massive parallel behaviour). We model each thread as a separate branch of a fork, where each branch gets the same amount of work. This is a valid assumption because the OpenMP parallel loop construct is implemented in the same way[3].

Therefore, depending on the number of threads needed, it is necessary to model not only one but *n* threads by *n* branches with *n* actions and divide the resource demand into equal shares. This process is labour-intensive and error-prone.

6.1.4. Experiment Evaluation

Table 6.1 shows the measurements and simulation results we collected by executing the program 500 times and by running the Palladio simulation. We computed the mean for both—the execution and simulation time. As one can see, the accuracy of the simulations drops when the number of worker threads (viz., the number of used cores) increases. One reason for the decreasing accuracy is that the simulation only considers CPU speed as a relevant metric, which leads to a linear speedup, while the measurements show that this is not the case.

[3] see OpenMP Specification: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`

There are several known reasons for not reaching an ideal speedup with a parallel program. We assume that the reason with the most significant impact is additional overhead created by the threads like synchronisation. In most cases, the matrices are not read or written directly from memory, but from the caches of the CPU cores. So, every time the result matrix is updated, the cache entries of other CPU cores become invalid and have to be synchronised or invalidated, which is expensive.

Regarding our research questions we summarise our findings:

Evaluation of RQ_{p1}: During the modelling phase, we show that modelling multicore systems is possible. However, a lot of manual and error-prone modelling was needed, since every thread had to be modelled individually. Finding ways to directly add parallel constructs (e.g., OpenMP parallel loop constructs) in Palladio is a desirable avenue for future research.

Evaluation of RQ_{p2}: To evaluate the accuracy of the simulation results, we performed several simulations according to the number of worker threads. We achieved the best accuracy for the sequential scenario, which is logical, since we used the measurements gained from the sequential run as calibration for the resource demand. But noteworthy is the decreasing accuracy as the number of worker threads increases. The decreasing accuracy indicates that predictions for even more worker threads will be even worse. That shows that further factors, like synchronisation overhead, have to be considered in the model to increase its accuracy.

Following the thesis hypothesis H_3 we assume that the inaccuracy is due to additional performance-influencing factors like cache sizes, memory size, and memory bandwidth, which are not considered in the model yet. The investigation of these factors follows in Chapter 7.

Having the result of the controlled experiment at hand, we can use it to define challenges and goals in the next section. Afterwards, we will use these goals to evaluate different modelling approaches of language extension strategies.

Worker Threads	Mean Execution Time (in s)	Speedup	Simulation Time (in s)	Accuracy (in %)
2	10.31	1.80	9.41	**91**
4	5.45	3.42	4.76	**87**
8	2.95	6.32	2.43	**82**
16	1.60	11.66	1.26	**79**

Table 6.1.: Simulations and Measurements Summary

6.2. Problem Specification - Challenges and Goals

In this section we use the insights from the controlled experiment described above and the lessons learned from the SLR (see Chapter 4). In the process, we identified challenges in modelling the behaviour of parallel software and the performance predictions for multicore systems. As the matrix multiplication use case makes clear, there are two major challenges [FKHB19]:

C_1 **Modelling Support for Parallel Constructs:** Current modelling languages like UML2 or PCM support concurrency aspects like threading. However, this means every thread must be modelled manually and in detail. Thus, the modelling is time-consuming and error-prone. Therefore, modelling languages have to support massive parallel executions of threads.

C_2 **Missing Overhead:** Even though the PCM supports simple concurrency aspects, the use case shows that their accuracy is a problem. The use case we looked at is easy to parallelise and only considers a limited number of cores (by now 128 is realistic). But also for 16 cores, the accuracy dropped by 21%. Besides many other issues (like missing performance metrics, e.g., memory architectures), all parallelisation paradigms produce additional overhead, e.g., for forking threads, synchronisation, and communication. Therefore, we need to find a way to include this behaviour in our models.

6.2.1. Goals

According to the challenges identified, we aim for the following goals [FKHB19]:

G_1 **Effort:** Reduce the modelling effort, so that the software architect does not need to model every thread individually.

G_2 **Language Constructs:** Similar to the OpenMP parallel loop construct, we aim for a single construct, which includes all relevant information. In this way the model is not inflated, and the complexity remains reasonable.

G_3 **Support:** All newly introduced concepts ease the modelling process, encourage understanding, and need to be designed such that the current tools for analysis and simulation can cope with them.

G_4 **Accuracy:** The prediction accuracy for parallel aware software components should increase without violating the G_1: Effort.

6.2.2. Evaluation Metrics

To be able to evaluate different language extension approaches, we define the following evaluation metrics [FKHB19], based on the goals identified in Sec 6.2.1:

E_1 **Configurable:** in terms of parametrisation and configuration. An approach is highly configurable if it enables the SA to easily change the model's configuration (i.e., thread numbers), and therefore offers the option to evaluate variations of configurations simply. A highly configurable approach is desirable.

E_2 **Additional Information:** describes the amount of additional information needed to use the language extensions. From the perspective of the SA it is desirable to add the minimum additional information required.

E_3 **Effort:** describes the amount of manual work. We distinguish the effort to inject an approach into a language (implement) and the effort to use the approach (use). A lower effort is desirable.

E_4 **Understandable:** means how intuitively one can use the approach. An approach is intuitively usable if (a) it can be used without much training and (b) the syntax supports the underlying semantics. A more understandable approach is desirable.

6.3. Modelling Language Extension

With the goals (G_1 to G_4) in mind, in this section we evaluate different variants to enrich existing modelling languages with parallel constructs. We first determine which different diagram types we consider relevant (see Section 6.3.1). Second, we propose different concepts (see Section 6.3.2) and third, we evaluate whether each combination of diagram type and concept meets the evaluation metrics E_1 to E_4 (see Section6.3.3). During the evaluation, we continue to use the running example, matrix multiplication, in combination with openMP, and regularly refer to it. Even though we use this specific example, we claim that the approach is transferable and works for different examples and parallelisation paradigms as well. We discuss that in the next section in detail.

6.3.1. Diagram Types

The PCM (see Section 2.4.2.1) provides different diagram types, which are candidates for an extension. We now have a closer look at the diagram types and their suitability for expressing parallel software behaviour:

(S)ervice (Ef)fect Speci(f)ication Diagram (UML activity-diagram-like): The SEFF is a suitable entry point for a modelling language extension since it directly describes the software behaviour. E.g., to describe the behaviour of the matrix multiplication, we use an `internal action` and a `loop action`. Therefore, the `loop action` or the `internal action` are potential extension points to define that either the loop or the action can be executed in parallel.

Repository Diagram (UML class-diagram-like): The repository diagram shows the available components in the system. Thus, one opportunity is to define a specific component and mark it as "parallel capable” and

set the whole component as parallel executable. That way two instances of the same component run in parallel, very much like in function-oriented architectures or micro-services.

Allocation Diagram (UML deployment-diagram-like): This diagram specifies which assembly (system diagram) is allocated to which resource container (resource environment diagram). Due to the deployment, the components are related to the hardware. At this step it becomes clear whether a component is running on a multicore system or not. However, no information about the software behaviour is available. Thus, the allocation diagram is not a suitable entry point for an extension.

Resource Diagram (UML component-diagram-like): The resource diagram only describes the hardware characteristics, so here we can express whether multicore CPUs are available or not. However, we can model no information about how the software utilises the cores and how the parallel behaviour takes effect. Thus, the resource diagram is not suitable.

Usage Diagram: Also in the usage diagram, no information about the parallel behaviour is modelled. Only information about user behaviour is available here. Therefore, this diagram type is not affected and not suitable for an extension.

6.3.2. Extension Concepts

Using a model means abstracting real-world objects and behaviour for a specific purpose [Sta73]. The challenge is finding the right level of abstraction as well as the relevant objects to represent in the model. In the following, we introduce three relevant elements (objects) for software characteristics, which are candidates to take into account while modelling the software behaviour. These concepts are independent of the above-described diagram types and can be included in any of them.

Overhead: The concept of overhead modelling considers overhead caused by parallel execution (i.e., thread initialisation, synchronisation, etc.). For example, if we parallelise a program using threads, the additional

overhead for creating, running, terminating, and synchronising the threads needs to be represented to adapt the speedup correctly.

Sequential Share: According to Amdahl's Law [HM08], a fraction of the software cannot be executed in parallel, which limits the speedup that can be reached by the software. Thus, the Sequential Share Modelling concept specifies the sequential parts in the models.

Shared Resources Behaviour: Using variables and resources in a parallel program is a challenge. In specific scenarios, it is essential that variables are not modified concurrently. Also, the program must modify the resources in the correct order. Further, where the resources are stored and how they are accessed is important. Using the Shared Resources Behaviour Model means considering this information on the model level.

Hybrid: Due to the characteristics of the concepts mentioned above, a combination of ideas is possible. In the following, we only consider the pure concepts, but we do not rule out the usage of multiple concepts later on.

6.3.3. Diagram and Concept Evaluation

Now that we know the relevant extension points (view types) and the possible concepts, we evaluate each combination based on the evaluation goals E_1 to E_4 (see. Section 6.2.2). Afterwards, we will take the combination which seems most promising, evaluate it based on the use case example, and propose it as a reference approach to create the parallel AT catalogue. This plan also means that we neglect the other combinations for now, but keep them in mind so that we can return to them if the chosen solution is not satisfactory.

The process to evaluate the combination is based on expert opinions. For this purpose, we conducted multiple review rounds within the Reliable Software Systems Group in Stuttgart, the Software Engineering Chair in Chemnitz, the Software Design and Quality Group in Karlsruhe, and with various external experts. The invited experts—mostly from German universities—work in different domains (Model-based Performance Prediction, HPC, Cloud Computing, and Parallel Programming).

Diagram	Eval. Goal	Concept		
		Overhead	Seq. Share	Shared Res.
SEFF	E_1	+	+	-
	E_2	o	o	-
	E_3	+	+	-
	E_4	+	+	-
Repository	E_1	+	+	-
	E_2	o	o	-
	E_3	-	-	-
	E_4	o	o	-
Allocation	E_1	o	o	-
	E_2	o	o	o
	E_3	-	-	-
	E_4	o	o	-

E_1 + easy to change — - hard to change
E_2 + no add. information needed — - a lot of add. information needed
E_3 + easy to realise — - hard to realise
E_4 + very intuitive to understand — - hard to understand

Table 6.2.: Summary for Different Extension Strategies

Table 6.2 summarises our evaluation and shows in the left column the three diagrams we selected as entry points. For each diagram type we used E_1 to E_4 as evaluation criteria (second column). The third to fifth columns show the three concepts, and an individual cell gives our final rating for a concept in combination with a diagram type based on the evaluation criteria. In the following, we will enter a detailed discussion.

Neglect the Allocation Diagram Even though the allocation diagram defines which component runs on which hardware, and therefore represents which component can run on a multicore system, using the allocation diagram seems unreasonable because at that point the definition of a component has already taken place. Parallelisation has to be enabled by software and therefore defined in the component description. Just because a component is allocated on a multicore system, does not necessarily mean it can be executed in parallel.

Neglect the Shared Resource Concept Handling shared resources by all kinds of parallelisation strategies is known as a complex and error-prone process. So, regardless of diagram type, including this concept will

require much effort to realise. But even then, the benefit is more than questionable, because to use the concept, the software architect needs detailed knowledge of the resources and their access, which should be abstracted during the design phase. Further, including a complex concept (like locks) into a design model dramatically decreases the understandability and increases the effort needed.

Evaluating the SEFF Diagram SEFFs represent the behaviour of components by, e.g., activity diagrams and therefore on a medium to low level. The concepts of loops and actions are known in these diagrams, and the structure follows the control flow. Reorganising often means only adding or removing activities or redirecting the control flow, and is therefore easy to realise. Because the abstraction level is not set for these diagrams, it is theoretically possible to model even low-level software behaviour. Therefore, concepts like overhead and sequential share could already be modelled with a lot of modelling effort (see Sec 6.1.1), and with a fair amount of additional information (i.e., thread pool size). However, without additional language constructs, the models became far too complicated and time-consuming to handle. Thus, the SEFF is a possible candidate for enhancement.

Evaluating the Repository Diagram The repository diagram represented as, e.g., UML2 Component Diagram, shows the composition of the components and therefore the architecture of the software on an abstract level. So if we want to integrate one of the concepts here, it must be on an abstract level as well. On the upside, that means changing configurations can be done quickly. On the downside, however, this means the understandability can suffer due to abstract representation. A fair amount of additional information is required, in this and in all other diagram types. But the effort to implement is high, due to the assumption that realising abstract concepts is always more challenging. Further, most parallelisation paradigms focus on low- to medium-level parallelism. Raising that concept to a higher level can result in inaccurate specifications.

6.3.4. Enhancement Process

Having the evaluation of the diagram types and the concepts at hand, in the following section we present the process for including parallel concepts (e.g., parallel loops) into a modelling language (like the PCM).

6.3.4.1. Choosing a Starting Point

After listing and evaluating all available options, we decide to focus on the SEFF Diagram with an overhead concept in the first run. Deciding for or against the repository diagram is a matter of abstraction level. While defining a component as parallel-capable means abstracting the parallelisation to the component level, and low abstract concepts like loops or section, focusing on loops means that the SA must already have an accurate idea of the software system during the design phase, which might not be the case. However, focusing first on the SEFF brings another advantage. The inclusion of the overhead model is better supported than in the repository diagram.

6.3.4.2. SEFF Language Extension

After choosing a concept and a diagram type, we now propose an approach to extend the language. This is a two-step approach: We first design a language construct to represent massive parallelism on the CPU level (like OpenMP parallel loops); and second, we add the overhead concept to the language to increase the prediction accuracy.

$Challenge_1$ **Modelling Aspect:** In the following, we focus on G_1 to G_3, which means we want to ease the modelling process for multi-threading and support parallel behaviour in the models. For proof of concept, we focus on the running example of the matrix multiplication in combination with OpenMP-like behaviour in our models. Since UML2 Activity Diagrams, as well as the PCM, already support loop-action, we focus on this action first.

The first question to answer is, which additional information is required to enrich a loop-action to a parallel loop-action. To answer that question,

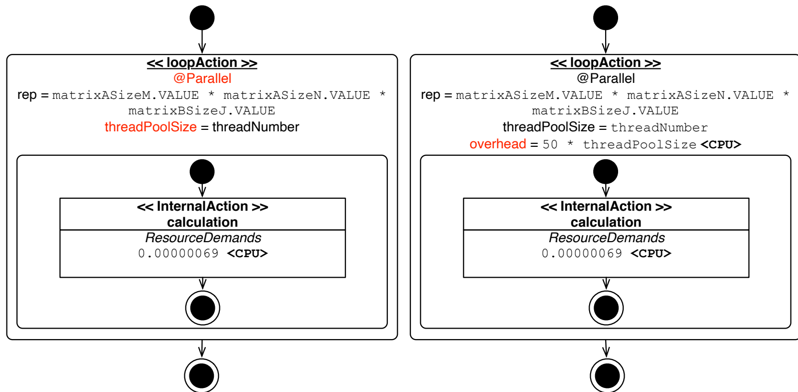
we follow the method for an experiment-based performance model derivation [Hap08]. According to this method, the performance model is extended in steps.

1. We identify a minimum set of additional attributes.
2. We add the additional attributes to the performance model.
3. We evaluate and check if the enhanced model fits the requirements.
4. If we see that it does not fit, or that we need different or additional attributes, we repeat steps two and three.

To identify the minimum set of attributes, we look again at the OpenMP parallel loop as a reference. As shown in Listing 6.1, the parallel loop only takes information about the number of worker threads used and the scheduler method (for a full discussion on performance-influencing factors see Section 7.2). Additionally, the scheduler method can already be set as a parameter of the CPU in the Resource Diagram of PCM. For the sake of simplicity, we start with the number of worker threads. Figure 6.3a shows the result of this first step.

Figure 6.3a shows a loop action annotated as parallel loop based on the PCM languages. There are only two differences to a regular loop action: The applied role `@Parallel`, which indicates that everything in the loop behaviour can be executed in parallel, and the number of worker threads attribute (`threadPoolSize`).

Challenge$_2$ **Accuracy Aspect:** In the following, we focus on G_4. For this, we decided to use the concept of overhead modelling first and include this concept in the PCM modelling language. To that end, we add the attribute to the parallel loop action from above. Figure 6.3 shows the parallel loop with the new overhead attribute. By allowing the attribute to be a dynamic value (as indicated by the sample value `50*threadPoolSize`), we can achieve two things at once. First, we enable the modelling of overhead, which can either be fixed or dynamic and equal for all threads (like thread initiation or synchronisation overhead). Second, we give the software architect the freedom to use this attribute to include a speedup function or, to be more precise, slow-down functions. For this, we allowed the specification of any



(a) Annotate Parallel Loop Including Thread Pool Size

(b) Annotate Parallel Loop Including Thread Pool Size and Overhead Function

Figure 6.3.: Stepwise extension of loop to a parallel loop

kind of stochastic expression (in PCM called stoex). In theory, this enables the software architect to model any type of behaviour here.

For clarity, in Figure 6.4 we show what a parallel loop would look like when using only existing concepts in PCM for a `threadPool- Size` of two. Figure 6.4 shows the instantiation of the parallel loop with two threads. It uses a fork action to fork two separate threads. Each thread has an internal action, which needs CPU-time. The resource demand is split equally among the two threads. Both threads are in a synchronisation point, which means they are synchronised after execution. In each thread, we add an internal action to describe the additional overhead.

6.3.4.3. Enhance the Modelling Language

Now that we have introduced the conceptual idea, we discuss in the following how the concept can be realised and integrated into existing models and analysis. First, we describe two different ways (Meta-Model Extension vs. UML Profiles) to extend modelling languages in general. Afterwards, we sketch the process of how to integrate them.

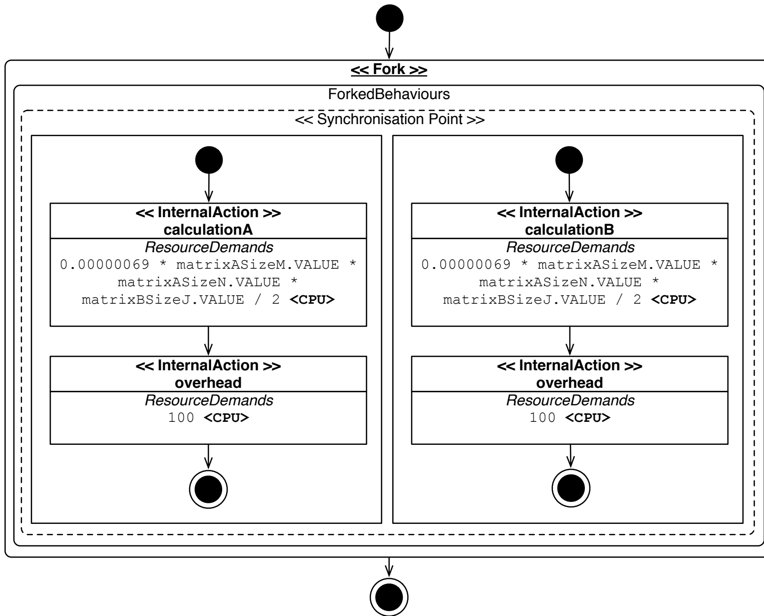


Figure 6.4.: SEFF Representation of the Unfolded the Parallel Loop Example from Figure 6.3

Architectural Templates & Meta Model Extension: There are two known ways to extend a modelling language like the PCM. The first way includes a full meta-model extension. In our case, this would mean extending the PCM directly and adding new meta-model elements and attributes.

The second approach is a profiling strategy. A UML Profile uses stereotypes and profiles to extend the meta-model without changing the actual meta-model. For the PCM there is a similar approach—the AT Method [LHB17]—which uses the AT Language. Within the AT Method, new language elements can be added, as long as there is a way to map the new language constructs to already-existing elements in the meta-model.

ATs vs. Meta Model Extension: With our scenario in mind, we identify the advantages and disadvantages of the inclusion strategies.

Using the AT method has many advantages. Once an AT is defined, it is easy and fast to use, and since every AT model extension has to be representable within the PCM, it is guaranteed that the simulations and analysing tools can handle the AT model extension. Thus we will not break any existing system. At the same time, this advantage becomes a disadvantage because mapping everything to existing meta-model elements also means limited power. Therefore, it might still be necessary to use a meta-model extension to achieve the intended outcome.

On the other hand, using a full meta-model extension is the most flexible option and gives us the freedom to integrate any kind of extension. However, this freedom comes at the cost of effort. Using a full meta-model extension means we would also have to adapt the performance prediction model and analysis tools to guarantee that the new language elements are supported.

In our case, we decided to use the AT method because it fits the use case best. As shown in Section 6.3.4.2, we are able to represent the new language extensions (see. Figure 6.3b) with the help of existing meta-model elements (see Figure 6.4). Note that other use cases may still require a full meta-model extension.

Architectural Template Extension Process: To use the AT method for our needs, we have to create a new AT. We can create a new AT by following three basic steps as described in [Leh18].

I. Create a Profile: First, we need to create a new profile. Creating a new profile is similar to creating UML2 Profiles. For our example this means we create a new stereotyped class called `ParallelLoopAction` and extend the target class from the PCM `LoopAction`. In so doing, we also model two attributes: `threadPoolSize` and `overhead` (see Figure 6.5).

II. Define Completion: In the second step, we need to define a model-to-model transformation. This is done with a QVT-o definition. The AT method contains a model checker, which is called before every performance analysis. Whenever the model checker finds an AT, it calls the QVT-o script and performs the model-to-model transformation to create a plain PCM model.

III. Register AT: In the last step, we have to add the newly created AT to the AT catalogue to make it available to the software architect via the Palladio tooling.

A full explanation of how the use case example is realised, along with a definition of additional relevant patterns, can be found in Section 6.5.

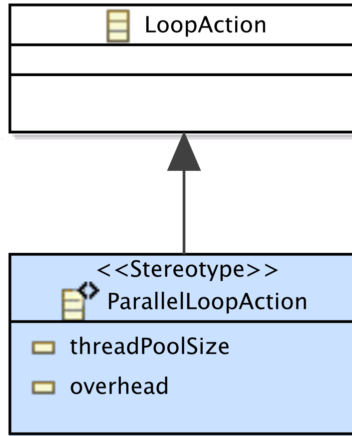


Figure 6.5.: AT Profile for Parallel Loop Extension

6.4. Proof of Concept Evaluation

In this section we present a proof of concept evaluation, using the running example. We apply the new parallel AT and evaluate it based on the simulation results, the predefined goals G_1 to G_4, and the evaluation metrics E_1 to E_4. If the evaluation is positive, we will use the approach to build a full parallel architectural template catalogue (see Section 6.5).

6.4.1. Result-based Evaluation

First of all, we evaluate the approach based on the prediction accuracy of the simulation. Here we use our use case example and compare the results of the altered model with the model we used in Section 6.1.3. More specifically, we use the newly introduced language extension concepts and remodel the use case using the new AT. So instead of using the `fork action` and modelling all the individual worker threads manually, we use a loop action and apply the parallel loop AT. Instead of creating different models for each number of worker threads, we were able to use the `threadPoolSize` attribute to configure the model.

The most challenging part, however, was to find a function to represent the overhead. For this evaluation, we want to keep the process of finding a good representation for the overhead as simple as possible. Thus we use the measurements we took from the implementation (see Table 6.1) for one, two, and four worker threads. We calculate the difference between a linear speedup and the actual measurements. Next, we extract a simple linear curve based on the number of threads as x and the difference of linear speedup and actual measurements as y. We ended up with the following equation, because it best fit the observations: `overhead = 900 - 50 * threadPoolSize`.

At first, this seems unnatural because we decrease the overhead while increasing the thread pool size. However, we increase the overhead per worker thread according to the workload while increasing the thread pool size. For two threads we have a total overhead of $1,600$ (800 for worker thread one plus 800 for worker thread two), and for four worker threads, we have a total overhead of $2,800$ (compare with Figure 6.3a).

Table 6.3 shows the simulation results when using a parallel loop action, configured as described above. The most noticeable outcome is that we achieve better accuracy in all cases. For one to eigth threads we reach 99 % precision.

The high precision is not surprising because we used the measurements from the real execution to calibrate the model. If we had used all measurments, we would have achieved an accuracy of 99% for all cases. This, however, would have been a model overfitting.

Worker Threads	Mean Execution Time (in s)	Speedup	Simulation Time (in s)	Accuracy (in %)
1	18.64	1.00	18.63	99
2	10.31	1.80	10.31	99
4	5.45	3.42	5.46	99
8	2.95	6.32	2.93	99
16	1.60	11.66	1.36	85

Table 6.3.: Simulations and Measurements Summary Using a Parallal-Loop-Action

Nevertheless, the evaluation shows us two things. First, the overhead modelling approach can be used to significantly increase the performance model prediction accuracy—if used correctly. Second, finding an overhead function, without having measurements from an implementation, is an extremely challenging task, which requires much experience in parallel computing and is still error-prone. Therefore, we propose to use characteristic performance curves to estimate the overhead function (see CB_4 in Chapter 7).

6.4.2. Goal-based Evaluation

In the next step, we evaluate the approach given the goal-fulfilment rate. We anticipate that we will reach all the goals G_1 to G_4. A detailed discussion follows:

G_1 Effort: Our first goal was to reduce the modelling effort so that it is no longer necessary to model every worker thread. In the proposed language extension, the software architect can just define the number of worker threads. Within the parallel loop AT, a completion is used to automatically generate the needed model and distribute the workload equally among all worker threads (as an OpenMP loop would do). However, this only works if the threads are identical. By definition, introducing automatisation reduces the overall effort.

G_2 Language Constructs: Our second goal—including all relevant information, to model parallel behaviour for performance predictions—is mostly fulfilled. We can prove that the goal is met by achieving a satisfying performance prediction accuracy. However, in the future, we need to discuss whether the level of abstraction is suitable. If not, it might become necessary to add further information or to abstract certain elements.

G_3 Support: The third goal required support for current simulators and analysis tools while reducing the modelling effort and keeping the complexity low. Since we decided against a meta-model extension and use profiling and stereotyping mechanisms, we are at least in theory able to use the full analysis support. However, currently, the AT approach is only supported by SimuLizar [Leh18]—the default simulator of Palladio. Moreover, the modelling effort is reduced by supporting the software architect with semi-automatic and model generation mechanisms. However, regarding clarity, we claim that the language extension did not increase complexity. We provide proof of this hypothesis in the empirical study (see Section 6.7).

G_4 Accuracy: Our fourth goal was to increase the prediction accuracy. If we compare Table 6.1 with Table 6.3 it became clear that we significantly improved accuracy for two to eight worker threads and also increased accuracy for 16 worker threads . With better overhead function, even better results are possible.

6.4.3. Metrics-based Evaluation

Finally, we evaluate the approach using the evaluation criteria E_1 to E_4. For this, we use the insights gained from the expert community. We consulted multiple experts from different German universities (e.g., TU Dresden - Department VDR and ZIH, TU Chemnitz - Department of Software Engineering and Operating Systems Group, HPI Potsdam, FZI Karlsruhe and KIT - Department of Software Design and Quality). In the following, we discuss the evaluation metrics in detail based on the results of the expert interviews:

E_1 Configurable: Due to the parameterizable character of the parallel loop extension, the approach is highly flexible and straightforward to

change. Therefore, the software architect can evaluate sets of configurations quickly.

E_2 Additional Information: As mentioned above, to use the parallel loop AT, the software architect has to specify two additional parameters. First is the number of worker threads, which is easy to set since a reasonable value can be the number of cores in the CPU. Second is the overhead function, which can be complex and hard to determine without detailed knowledge of the program and parallel computing in general.

E_3 Effort: Regarding the effort of integrating the approach, we cannot give a long-term answer because a comparison is missing (see Section 6.7). However, as described in 6.3.4.3, using the AT approach has the advantage that the meta-model does not need to be changed, and therefore all analysis support is still guaranteed. Further, using an AT eases the modelling process for a software architect and reduces the effort in general—as described in [LHB17].

E_4 Understandable: The usage and clarity of ATs in general is also discussed in [LHB17],but for our specific use case, we cannot provide a definite statement. However, all of the experts interviewed agree that the approach can be used without training—assuming the software architect knows how to use ATs—and the underlying semantics are implicit in the syntax. The empirical evaluation of the architectural template catalogue, however, proves that even non-experts can use and apply ATs correctly.

6.5. Building a Pattern Catalogue

After evaluating the above approach, we will use this approach in the upcoming section to create a parallel architectural template catalogue, containing the parallel behaviour patterns most often needed by software architects. In the first part of the section, we focus on the research, collection, and identification of such relevant patterns. In the next section, we give a detailed behaviour description for each pattern. Finally, we will visualise the empirical study we used to evaluate the usability of the template catalogue.

We will not further discuss the implementation details of the individual patterns, but we will follow the approach described above. The full pattern catalogue, along with the source code and further documentation of the individual patterns, is available in the parallel AT catalogue repository on GitHub[4].

6.5.1. Pattern Identification

The first question we have to ask when building a pattern catalogue is: which are the relevant patterns? To answer this question, we formulate two sub-research questions: ($RQ_{1.1.1}$) Which parallel patterns already exist in practice and ($RQ_{1.1.2}$) do they have similarities which allow them to be categorised? To answer that question, we performed a structured literature review in [SWD19]. The results of this study are presented in the course of this section.

6.5.1.1. Search Method

To answer the $RQ_{1.1.1}$, we performed a structured literature review and followed this process:

1. **Initial Set:** We start the structured literature review by building an initial set of parallel programming patterns that we already know. We took most of these patterns from [MSM04].
2. **Searching:** In the next step, we used the initial set to query four different databases: ACM, IEEE, ScienceDirect and Google Scholar. We rejected duplicates.
3. **Screening:** In the next step, we started from the top of the list and screened each hit and then extracted pattern names and description. If a pattern was already in our list, we ignored it.

[4] `https://github.com/PalladioSimulator/Palladio-Addons-ParallelPerformanceCatalogue`

4. Abortion: Due to a large number of search results, we decided to continue step 3 until we encountered 20 consecutive papers with no new patterns. The danger of this approach is that we most likely will not find all existing patterns. However, we can be quite certain that we cover the most relevant ones. This is good enough for building a first version of a parallel architectural template catalogue. Extending additional patterns later will not require much overhead.

After conducting the search, we ended up with 35 patterns. As we had assumed, many of them follow the same concept but are named differently.

6.5.1.2. Pattern description

In the following, we give a short overview and a brief description of the 35 patterns we found, as reported in [SWD19]:

"Actors: Actors is a distributed parallel approach using a message-passing interface. Actors communicate by sending messages that determine the workflow. We took a detailed look at the Actors approach for its message-passing approach to parallelism. Details about this approach can be found in section 2.1.3.

Fork/Join: The Fork/Join approach is a shared memory approach that divides a problem over a certain size into smaller sub-problems, which then compute the smaller tasks in parallel. After a parallel computation step is finished, the split tasks are joined. Details about this approach can be found in section 2.1.3.

Parallel Loops: Parallel Loops is an approach used on index sets. By changing how the set is iterated, and in so doing, splitting the set into smaller parts (i.e., only even/odd indices), thread-based parallelism is achieved with little overhead. Thus it uses shared memory. Details about this approach can be found in section 2.1.3.

Pipes & Filters: The Pipes & Filters approach uses components called filters that are connected by pipes. Filters can be used in parallel to speed up the computation of high load-bearing tasks. It is a very modular shared memory approach and thus distinguishes itself from other patterns. Details about this approach can be found in section 2.1.3.

Master Worker: A master thread is used to generate several worker threads, each capable of recursively becoming a master thread. Worker threads are assigned tasks on shared memory by the master and report back upon finishing a task to collect results. This pattern is very similar and follows the same concept as the fork/join pattern.

Java Streams: Java Stream is a parallel approach that splits a collection of shared memory into several streams and applies an ordered set of operations on each stream, and finishes by merging the streams into a new, transformed collection. It is an application of the Pipes & Filters approach [Mic19].

SPMD: Single Program Multiple Data. A set of Unit of Execution (UE)s running the same algorithms on different subsets of data is decomposed from an initial shared set of data. It can, for example, be used to implement a Master Worker approach. It is not a pattern itself, but a supporting structure [MSM04, p.216].

MPMD: Multiple Programs Multiple Data. This is like SPMD, but each subproblem of the initial shared data gets mapped onto a subset of the unit of executions running the algorithms needed. It is not a pattern, but a supporting structure [MSM04, p.216].

Map-Reduce: Map-Reduce allows a mapping procedure similar to a Stream's intermediate operations on a set of shared memory to work as a filter/sort. After a set of mapping procedures, a summary operation (reduce) will finish the task. For our purposes, a Map-Reduce approach operates similarly enough to Parallel Streams that it does not need to be described in detail [DG04].

Akka Actors: The Akka Actors is an implementation of the Actors model using the Akka libraries, which allow for writing concurrent and distributed systems. The Akka approach to actors is an instance of the generic Actor model, and as such is not different enough to be considered a separate approach [HBS73].

Erlang Actors: The Erlang Actors is an implementation of the Actors model using the functional programming language Erlang. Unlike Akka, it does not rely on sending messages as objects, but the overall implementation of the Erlang Actors approach makes this another instance

of the generic Actor model, and as such will not be considered an approach.

Task Parallelism: Task Parallelism is a concept rather than a pattern. It focuses on distributing tasks (a set of operations) over multiple UEs with the intent of calculating multiple tasks on the same shared memory at the same time [MSM04, p.67].

Data Parallelism: Similar to Task Parallelism, Data Parallelism involves running different sets of data on UEs with the same tasks. It, too, is more of a concept than a pattern [MRR12, p.372].

Divide and Conquer: A concept that involves dividing an initial problem into a set of subproblems before the computation is known as Divide and Conquer. Is not a parallel pattern by itself [MSM04, p.76], but a method of implementing parallelism.

Geometric Decomposition: Geometric Decomposition divides a set of data not into a set of subproblems, but rather into chunks of regionally close data such as one finds in graphs. Similar to Divide and Conquer, it is by itself also not a parallel pattern [MSM04, p.82].

Recursive Data: Along with Divide and Conquer and Geometric Decomposition, Recursive Data is also not a pattern by itself, but a way of dividing a set of data into subsets. It is especially useful for recursive sets of data [MSM04, p.101].

The following patterns are found in [MRR12] and are duplicates of already named patterns:

Nesting Patterns: Nesting Patterns is a compositional approach that describes a method of composing code using several approaches. It is not a pattern in itself, but is applicable to most approaches.

Map Pattern: The Map Pattern applies a function using loops on every element of a set of data A with a resulting set of data A’. It is used with index sets and can be used on a single UE or on multiple UEs. On multiple UEs it becomes an instance of the Parallel Loops approach and as such will not be discussed.

Stencil: The Stencil approach, similar to the Map Pattern, applies a function on an index set, but instead of only working at single elements, it also looks at neighbours of each index. As a variation of Map Pattern, it is not considered distinct enough for our purpose.

Reduction: Not a pattern but a method of combining all elements in a collection into a single element, Reduction can be executed in parallel.

Scan: Scan is similar to Reduction, but every step of the reduction produces a new element that adds up to the partial reductions that were calculated in the steps before. Not a pattern but a data management method, it can be executed in parallel.

Recurrence: A specialisation of the Map and Stencil approaches, where outputs of neighbouring indices can be used as additional input and used in cases where elements of a set are not independent. A parallel implementation of Recurrence also becomes an instance of the Parallel Loops approach.

Superscalar Sequence: An approach where a serial sequence of tasks is not dependent on order apart from data dependency, and as such can be executed in a random sequence or in parallel. In theory, this concept is practised in many parallel approaches.

Futures: Futures are a Fork/Join approach using heaps instead of stacks.

Workpile: Workpile is a modification of the Map pattern. Each visited element can generate new tasks that are added to the index set, allowing recursive behaviour. It is very similar to parallel loops or sections.

Pack: The Pack approach is used to reduce the size of collections by mapping unneeded values to zero. A data management approach, it is an instance of a Map-Reduce approach.

Expand: Expand is similar to the Pack approach, but each element of a collection can output any number of elements including zero. It is a subset of the Map-Reduce approach.

Search: The Search approach finds data in a shared set that fits a given criterion. It is considered a function that is part of the mapping process in the Map-Reduce approach.

Category Reduction: Category Reduction collects elements in a labeled collection using a map function and reduces them to their categories. It is an instance of a Map-Reduce approach.

Gather: Gather reads a sub-collection of data from another collection of data. It is considered a map pattern, but works with "position" rather than values inside of elements in a collection. The Stencil approach uses this method to acquire a neighbourhood of values.

Scatter: Scatter is similar to Gather, but the input set of data is written to a set of specified write locations in parallel. Multiple variations exist to deal with collisions. This is a data management function and not a pattern in itself, and is too specialised to be part of this research." ([SWD19], p.14-17)

The remaining four hits are SISD, SIMD, MIMD, and MISD and these are not software behaviour patterns but hardware architecture styles. Thus we ignore them as we build the taxonomy.

6.5.2. Pattern Categorisation

After collecting patterns and extracting characteristics, we went through the result set again and started to group similar patterns. We named each group according to the most common name and also introduced an additional dimension, the abstraction level. We added three levels of abstraction: Algorithmic, Architectural, and Design Patterns. Figure 6.6 shows the result by grouping architectural and design patterns for simplification. For each pattern, Figure 6.6 lists synonyms or implementation variants based on the findings of the structured literature review. This list is not complete and provides only an overview.

For a detailed explanation of the individual groups, see Section 2.3.

6.5.3. Pattern Selection

After we successfully categorised all patterns, we extracted the core behaviour from each group of patterns. For three out of the four groups, we decided to realise a parallel AT, but decided against the message-passing

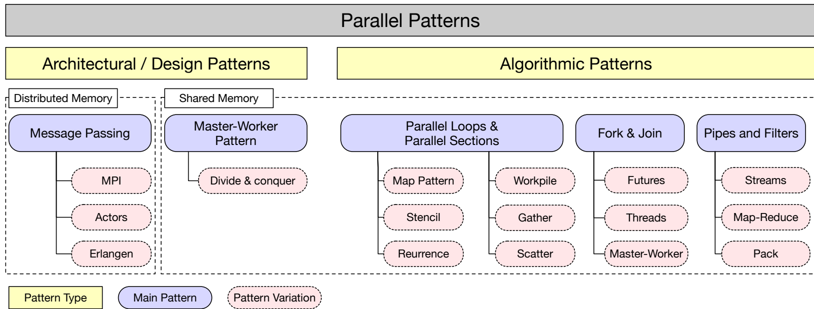


Figure 6.6.: Categorisation of Parallel Patterns

pattern, for several reasons. The message-passing paradigm follows a different concept and assumptions, which are fundamentally different from the other three patterns, as well as the concept Palladio is based on (especially as represented in Actors). Palladio builds upon the assumption of passive and stateless components. However, an actor is a state-full and active component. Ignoring this fact will lead to a violation of the Markovian properties, which the Palladio simulations and analyses are based on. Therefore, we decided against a realisation of the message-passing paradigm in an AT [SWD19].

For all the other patterns, we followed the proposed approach and realised a corresponding parallel AT. We published the complete parallel pattern catalogue along with the source code in a Palladio sub-repository on GitHub[5].

6.6. Formal Semantics for Parallel Behaviour in the PCM

To create or use parallel modelling language elements, it is crucial to understand the semantics of their behaviour. Therefore, in the course of this section we will explain the semantics of the most relevant parallel languages elements in the PCM and the semantics of the parallel ATs. To do so, we will use a formal specification with the help of HQPNs (see Section 2.5).

We start by explaining the mapping of fundamental PCM components to Hierarchical Queuing Petri Nets (HQPN), which was developed by Koziolek

[5] https://github.com/PalladioSimulator/Palladio-Addons-ParallelPerformanceCatalogue

in [Koz08]. Koziolek defined semantic behaviour for most of the PCM elements. However, we will discuss only the loop and asynchronous fork at this point, which we later reuse for our parallel ATs. For a full definition of all fundamental elements of the PCM, we refer to Koziolek's dissertation [Koz08].

Second, we introduce a mapping for asynchronous loops, which was not done by Koziolek.

Third, we discuss mapping the parallel behaviour to QPNs in general. Based on that, we will evaluate and compare the semantic behaviour of the parallel ATs (from [FH18]) to the expected parallel behaviour.

6.6.1. Mapping of general PCM Components

All elements used in the following are part of the Palladio SEFF, which describes the behaviour of the software model. For the sake of simplicity, we only use subnets (QPN) of the HQPN.

Within our HQPN each token represents a single user or request within our system. The token's colour is a complex data type named `TokenData` (see Lst. 6.2). It contains:

- **VarList**: A list of currently valid parameter characterisations.
- **CompParList**: A list of currently valid parameter characterisations specified as component parameters.
- **LoopList**: A list of loop iterations. When a token enters a loop, the loop iteration number is set in the list to show the number of iterations that remain.
- **GuardList**: A list of branching guards. The PN uses them to calculate probability distributions with stochastic dependencies.
- **TokenID**: A unique ID for each token. The ID can be used to merge tokens after they have been split and fire them into subnets.

In the following we adhere to Koziolek's semantics and refer to the `TokenList` as `a`. For further details on mapping the processing resources, stochastic expressions, and distributions, see [Koz08].

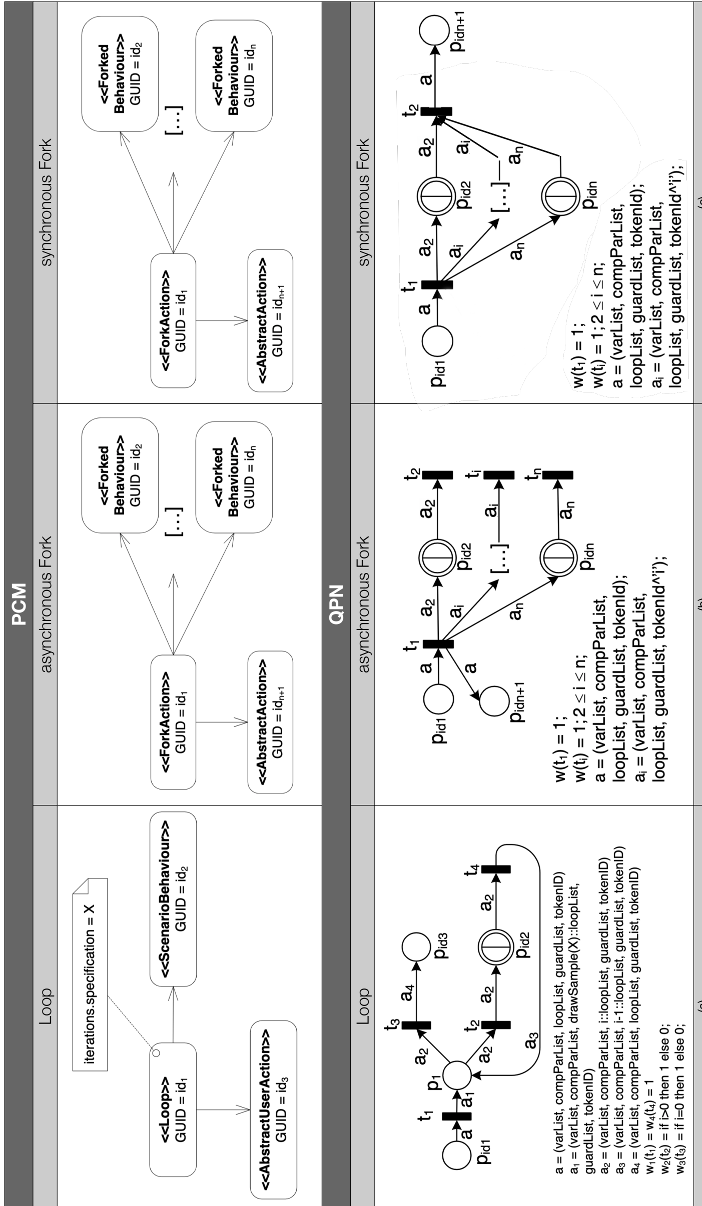


Figure 6.7.: Mapping of PCM2QPN: (a) LoopAction, (b) Asynchronous Fork, (c) Synchronous Fork (for a,b cf. [Koz08])

```
color VarSpec = product string * string;
color VarList = list VarSpec;
color CompParList = list VarSpec;
color LoopList = list int;
color GuardList = list string;
color TokenId = int;
color TokenData = product VarList * CompParList *
    LoopList * GuardList * TokenId;
```

Listing 6.2: Colour of a token, called TokenData (cf. [Koz08])

6.6.1.1. PCM Loop

Figure 6.7a shows the mapping of a PCM Loop Component (on the top as PCM description) to a QPN (below). The QPN contains the loop head and body. After entering the loop, the first transition t_1 is to evaluate the loop iteration (in case it is not a constant value, but a distribution or stochastic expression). The transition t_1 adds the loop iteration integer as a list instead of an integer to the `LoopList`. The reason for this is that the loop can be executed recursively nested, and the token needs to memorise all the loop counters. The head of the list gives the current iteration count.

Based on that value, either transition t_3 (counter = 0) or t_2 (counter > 0) fires. If t_2 fires, the token will be fired in a subnet p_{id2}, which represents the loop body. As soon as the token returns from the subnet, t_4 fires, a_3 decreases the loop counter, and the token enters the loop head. Finally, when t_3 is reached, a_4 removes the counter from the list of loop iteration integers and the token is placed in the successor of the loop (i.e., p_{id3}).

6.6.1.2. PCM Asynchronous Fork

Asynchronous Forks spawn new threads without synchronising them in the end. Each thread terminates independently of the others. Figure 6.7b illustrates the behaviour for the given PCM specification (above).

First, the transition t_1 fires a copy of the current token into multiple places in QPN p_{idi}, each representing a forked behaviour. During t_1, the values of the current token are modified in a way that the ID h stays unique. For that,

a number i is added for each forked behaviour. The rest of the values stay the same. At the end of each forked behaviour, the transition t_2 - t_n flushes the copied token. To continue, the transition t_1 fires an additional token to the successor, represented here by p_{idn+1}.

6.6.1.3. PCM Synchronous Fork

In contrast to asynchronous forks, in synchronous forks the control flow spawns threads and waits for them to finish before continuing with the next steps. Figure 6.7b illustrates the behaviour and describes the PCM.

In general, the QPN looks very similar to the asynchronous forks, so in the following, we only go into the two main differences.

First, instead of the transition t_2 to t_n (in asynchronous forks), which flushes the token after the forked behaviour has finished, for synchronous forks we have one transition t_2, which only fires if there is a token available in each place p_{id2} to p_{idn}. If that is true, t_2 fires and places a token in the successor of the synchronous fork—in our case p_{idn+1}. The token that is placed in the successor place is a merged copy of a_2 to a_n. Further, the ID h is modified so that i is removed. Thus the ID is reset to the original value before entering the fork, and remains unique.

The second difference to the asynchronous forks is when and how to pass the token to the successor. While for the asynchronous forks, the transition t_1 immediately passes a token to the successor, the transition in the synchronous forks does not and only passes the tokens into the forked behaviours. The successor is added in the end, and the transition t_2 triggers the successor. In that way, we ensure that all forked behaviours have finished before continuing.

6.6.2. Mapping of Parallel Behaviour to QPN

In this section, we discuss the behaviour of parallel loops, sections, and blocks. Since no native PCM elements represent these concepts, we give the PCM descriptions based on the parallel AT extensions introduced above.

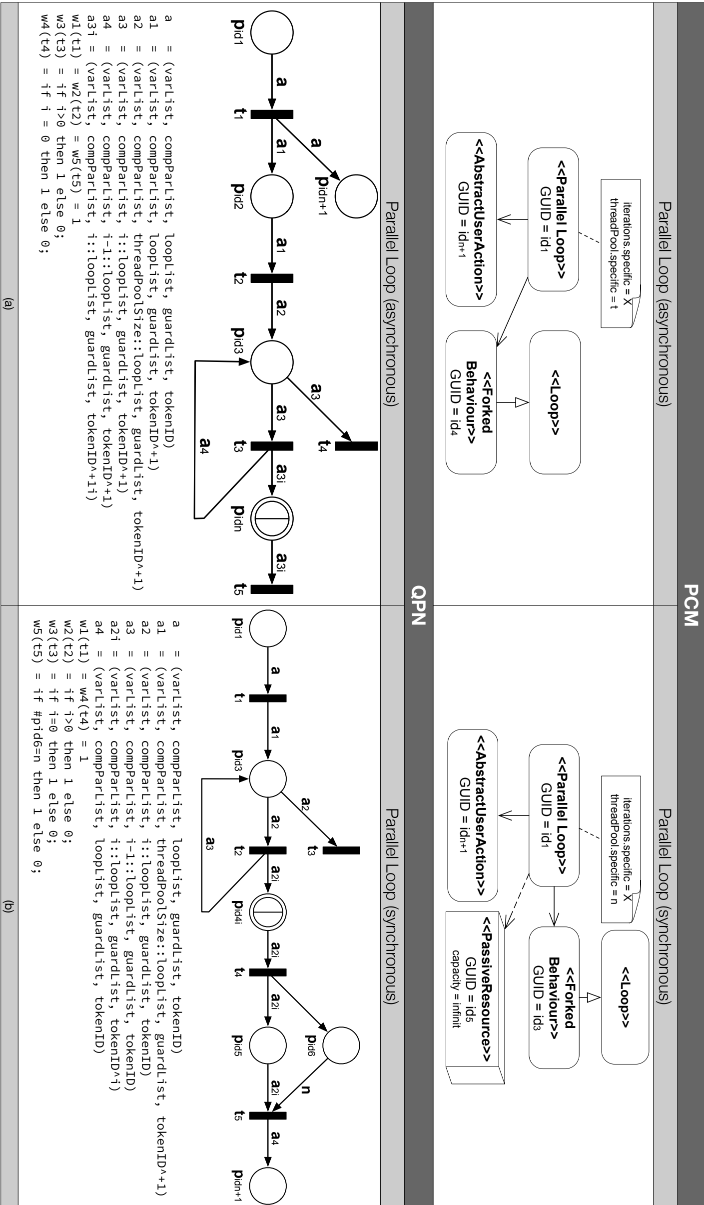


Figure 6.8.: Mapping PCM2QPN: (a) asynchronous parallel loop, (b) synchronous parallel loop

The description should reflect the way common frameworks like openMP[6] have implemented these concepts.

6.6.2.1. Parallel Loops

Behaviour: Parallel loops are a parallelisation concept known from different parallel programming paradigms like OpenMP. Put simply, a parallel loop executes each loop iteration in a separate thread. With the help of a thread pool, the scheduler assigns each thread (worker thread) to a physical core and can execute in parallel. A requirement for the many scenarios is that the threads are data independent or that the dependence is explicitly defined. Data independent means that the read and write operation of each thread does not influence the others. A typical example to illustrate the behaviour of parallel loops is our running instance of a matrix multiplication [FH16]. Assuming we have two matrices (10x10) we want to multiply, this would result in a total number of 1000 multiplications to perform. Using, for example, OpenMP parallel loops with a thread pool size of 8, this would split the workload for each thread equally, resulting in 125 calculations per thread.

A parallel loop can either be synchronous (often used when distributing workloads and realising a master-worker pattern [MSM04]) or asynchronous (i.e., implementing an observer pattern).

PCM Instance: Given the above behaviour description of a parallel loop, it is similar to a fork action in PCM. It has a successor and a forked behaviour. Since the behaviours are all equal, specifying it once is enough. In addition to the fork action, information about the thread pool size and the number of iterations is required. For synchronous forks, a passive resource is needed as well. A passive resource can be used to implement require and release behaviours, i.e., for mutexes [Koz08].

Mapping: For the mapping of the behaviour description to QPN, we distinguish between two different kinds of parallel loops: Synchronous and asynchronous loops, which are shown in Figure 6.8.

[6] OpenMP – https://www.openmp.org/

Asynchronous Parallel Loop: The QPN for asynchronous parallel loops is a combination of a loop and an asynchronous fork. It starts similarly to a fork with the transition t_1, which fires two tokens. One token is fired in the place of the successor p_{idn+1}, which can then continue, and another token is fired into the place of the loop behaviour. The id of the token is altered and increased (a_1). Following the description of a loop (see Figure 6.7a), the next step evaluates the loop iteration. In this case, two evaluations are done. One is for the outer loop, which forks the new threads. Here the value equals the value of the given thread pool size. The evaluation of the iteration literal specifies the second loop iteration value and then divides it by the thread pool size, to share the workload equally. It is added to the `LoopList`. Based on that former value, the loop either continues or finally goes to t_4. If the loop continues, t_3 fires two tokens, one into the subnet p_{idn}, with an adjusted id (cf. Section 6.6.1.2), and one to p_{id3} with an adjusted loop counter. After that, the loop condition is re-evaluated. Further, the subnet p_{idn} represents a normal loop as characterised in Section 6.6.1.1. Finally, when a subnet has finished, t_5 destroys the token.

Synchronous Parallel Loop: In contrast to the asynchronous parallel loop, the synchronous one does not continue until all tokens have returned from all subnets. For that reason, there is no fork action in the beginning, and the QPN starts with the evaluation of the loop iteration, which again equals the value for the thread pool size. The loop execution behaves the same way as the asynchronous loop does. In contrast to asynchronous loops, where tokens are flushed after returning from subnets, in the synchronous loop the tokens are passed on. The transition t_4 fires a token into two places: p_{id5} and p_{id6}. Further, p_{id5} shows a passive resource and X indicates the number of created tokens. Therefore, whenever a subnet finishes and the token returns, t_4 fires and increases the number of tokens in the places. Subsequently, the original token with the corresponding colour is placed in the p_{id5}, and the loop iteration counter is removed from the token's colour. Finally, transition t_5 fires if there are the number of n tokens in the place p_{id6}. The value of n is equal to the value of the thread pool size. Thus, the transition t_5 fires if all subnets have been returned. Further, the transition t_5 adjusts the value of the id field, removes the added identifier for the subnet i, and restores the value to its original value.

Please note that to provide a useful example, we modelled the passive resource (p_{id6}) along with the require (x) and release (n) actions explicitly. It is also possible to combine it with p_{id5}.

6.6.2.2. Parallel Sections and Blocks

Behaviour: Parallel sections or blocks refer to a specific area in the source code that is either explicitly marked for parallel execution (i.e., parallel sections in OpenMP) or implicitly allows multiple executions of the same block. The former behaves similarly to a loop. Most of the time, a parallel section is used to split the workload based on a task set or data structure. The block is specified by the same behaviour, but can have different input parameters. It can be a method that is called by multiple threads.

PCM Instance: In the PCM a block, which can be called multiple times from different threads, is modelled with a simple fork action and therefore can be either synchronous or asynchronous. Due to the similarities of a parallel section to a parallel loop, there is no additional concept in PCM, and on an abstract level, it can be handled in the same way as a parallel loop.

Mapping: The mapping of PCM Instances for parallel sections to QPN is performed in a way very similar to the mapping of parallel loops. The only difference is that the subnet will not be of type loop, but arbitrary types. This means that it is not the loop characterisation that is passed to the subnet, but an adjusted version of the `VarList`, describing the workload for the specific subnet. For blocks, the mapping is the same as for forks. Due to these highly similar concepts, we will skip a full description at this point.

6.6.3. Evaluation of the Mapping of Parallel ATs to QPN

In the following, we evaluate the correctness of the behaviour of the parallel loop ATs based on the running example. As described in Section 6.5, the parallel ATs need to map all elements to the given PCM instances. Since loops, sections, and blocks are very similar, the parallel AT method maps all kinds of parallel behaviour (loops, sections, or blocks) to a fork-join

scenario (see Figure 6.3). Therefore, we can use the existing mapping of forks to QPNs to express the formal semantics. To show that this is a valid approach, we elaborate on a thought experiment. For that, we assume a synchronous parallel loop, which should calculate a matrix multiplication with the matrices of size $10x10$. So, in total 1000 multiplications have to be performed. Further, we assume each multiplication takes $1ms$ on a two-core system. In theory, sequentially executing the multiplication takes $1s$. Using a synchronous parallel loop (as described in Section 6.6.2) needs additional information about the number of worker threads. Assume we use two worker threads for the two-core system. The behaviour of the synchronous loop splits into two separate threads, which share the workload equally. That means each worker thread needs to perform 500 multiplications and needs $500ms$. Since we assume two cores, the overall execution time is $500ms$, because both threads can run in parallel. Now let us consider the parallel AT: Here we use the parallel loop action (see Figure 6.3a) and specify the number of replications to be 1000, the thread pool size is two, and the resource demand for one calculation is $1ms$ on the CPU. The parallel AT approach now maps this to a fork behaviour with two parallel threads, which needs to be synchronised in the end. The resource demand for each internal action is still the same $1ms$ on the CPU. But this time, it is multiplied by the number of repetitions divided by the number of worker threads (i.e., it shares the workload equally). In this case, each internal action takes $500ms$, and the total run-time is $500ms$.

This demonstrates that the response time behaviour is the same. For this, in future work, we plan to provide mathematical proof based on QPNs.

6.6.4. Upshot

In this section, we formally defined the semantic behaviour of the fundamental parallel language concepts fork and parallel loop. This will not only help to create and use new parallel language concepts, but it also helps to understand the parallel ATs. At this point, we only explain fork and parallel loop, since the other two parallel ATs—Master-Worker-Pattern and Pipes and Filters—are mapped and build upon the same basic constructs as the parallel loop.

6.7. Empirical Evaluation of the Parallel AT Catalogue

Now that we have all the parts of a parallel AT catalogue complete (process to enhance modelling languages, pattern selection, behaviour descriptions, and finally the catalogue itself), we still need to evaluate $RQ_{1.3}$—Does the architectural template catalogue support software architects in the task to create accurate performance prediction models efficiently?

We have already shown how we can use an overhead function to increase accuracy. In this section, we want to evaluate the efficiency and usability of the approach. Since both quality aspects are hard to determine, we set up an empirical user study. This study was part of the work we conducted in [Zah20], and we present a summary in the following subsection.

6.7.1. Experiment Design

To conduct a user study, we decided to go with a controlled user experiment. The controlled experiment gives us the advantage of minimising variance and disturbing side effects and gives us the opportunity to change the experiment variables according to our needs [RH09]. Further, it allows us to perform statistical analyses on our measurements [WRH+12]. To determine and specify the necessary metrics, we use a Goal-Question-Metric (GQM) [CR94] plan to define goals, questions, and metrics.

In total we derive four goals from the given $RQ_{1.3}$. Figure 6.9 shows the GQM-tree.

For each goal, we formulate the corresponding question, the metric we want to measure to answer the question, and the hypotheses we have regarding the outcome. With questions two to four, we would like to determine which metrics to measure during the user study. So we measure the time participants will need to fulfil a task, the number of errors they make, and the time they need to fix mistakes. In contrast to that, we answer question one by evaluating a questionnaire that each participant completes.

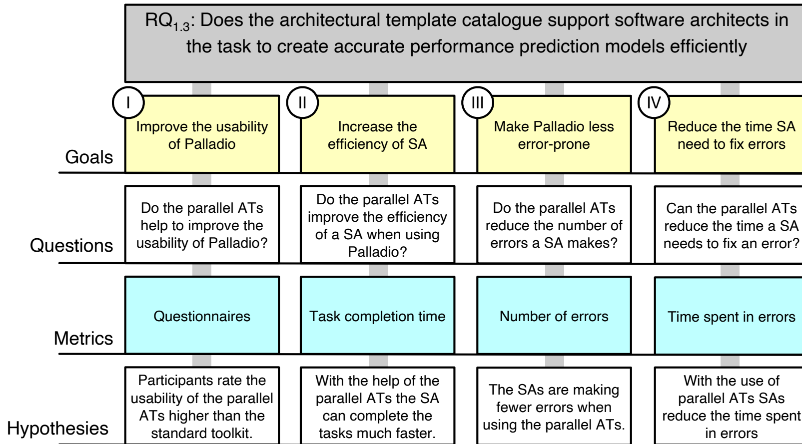


Figure 6.9.: Goals, Hypotheses, Questions, and Metrics of the User Study

6.7.1.1. Conduction Process

Given the above GQM-Plan, we developed an experiment design and study process. Figure 6.10 shows the experiment design. It contains three phases:

Phase 0 – Warm-up: During this phase, we first want to recruit participants. To get the most reliable results, we aim to have a mix of diverse participants. Their experience with performance engineering should range from none to expert. Finding experts will be more difficult since they are rare. However, if we can show that beginners using the parallel AT catalogue are better (in terms of the above questions) than experts who are not using the parallel AT catalogue, we can make a strong statement, even with only a moderate sample size.

The next step during warm-up is to train the participants. During this step, we will teach each participant the requirements to fulfil the task, as well as educate them on the tool we want to use. Since we do not want to measure how well participants can learn new tools, we do not monitor this step in any way. However, we provide feedback, answer questions, and ensure that all participants complete the training.

The last step is to split the participants into two test groups. Both groups should be equal in size and experience level. Hence, each group should contain the same number of experts, advanced users, and beginners.

Phase 1: In the first phase, the participants are assigned to groups and scenarios. Each group has to complete the same scenarios. However, the order in which they should use the parallel AT catalogue differs. Group A needs to complete scenario I with the standard toolkit, while group B uses the parallel AT catalogue to do so.

During the execution of scenario I we measure the overall time, the number of errors, and the time each participant spends on errors (Appendix A.3 shows the sheet we use to take the measurements). After completing the task, each participant has to fill out a questionnaire (see Section 6.7.1.3).

Phase 2: The last phase is similar to the first one. This time the participants get a second scenario, and we switch tasks for the groups. Thus, group A has to use the parallel AT catalogue and group B the standard toolkit. This way we can rule out any learning effects participants may show during the completion of the first scenario. We again measure the times and errors. Afterwards, participants have to fill out a questionnaire again, and finally, we interview them.

6.7.1.2. Scenario Selection

In addition to the above-formulated GQM-Plan and process, we also need a scenario. The scenario will be presented to the participants, and they will have to solve the task afterwards.

The first scenario involves the running example of the matrix multiplications and is fully described in Appendix A.3 Scenario II.

The second scenario describes a parallel search strategy to find literature in a literature database (see Appendix A.3 Scenario 1 for a full description).

Both scenarios have in common that they need to fork multiple threads that are performing a similar task. For each thread, there is some overhead for

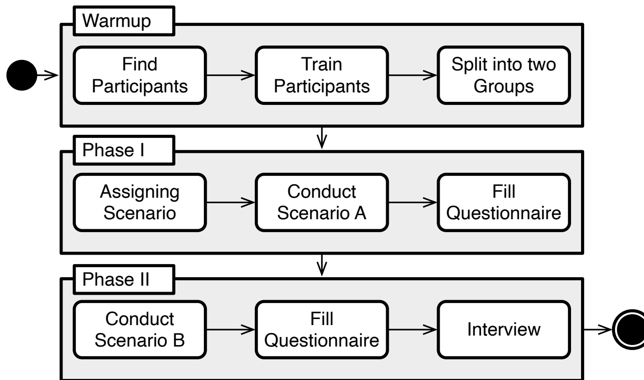


Figure 6.10.: Overview of the User Study

forking and synchronisation. Besides that, the threads are independent of each other.

We ensure that both scenarios could be modelled in Palladio with and without the parallel AT catalogue.

6.7.1.3. Questionnaire

To capture general information about the participants and to rate the usability of the parallel AT catalogue, we design a three-part questionnaire that each participant has to fill out. In the first part, we ask for general information about the participant, like their current degree, their level of expertise with performance engineering, and their experience with Palladio. Based on this information we design the user groups A and B and aim for a balanced group.

The second part contains four short questions, which have to be filled twice by the participants—once after each scenario. Here we ask about the difficulty of the scenario, how they would rate their own performance, the amount of work they had to do, and how they would rate the usability of the standard toolkit/parallel AT catalogue for the scenario.

The third part contains a total of six questions. The first three are about the usability and speed of the parallel AT catalogue in comparison to the standard toolkit. In the first three the participants are asked to use a scale from one to seven. The latter three are free text fields, where the participants give their final thoughts about general aspects of the experiment. Appendix A.3 shows the full experiment leaflet with all questions, scenario descriptions, and information provided to the participants.

6.7.1.4. Analysis Process

To answer questions two to four, we can consider the measurements we took off time and number of errors during the experiments. However, to answer question one, on usability, we have to consider participant feedback. In the questionnaire, the participants can rate the usability of different items, using a scale divided into seven levels. We can now translate the levels in a numerical schema ranging from one to seven. For each question, we calculate the mean value.

Now that we have numerical values for all questions and thus our metrics for the hypothesis and goals, we can directly analyse some of them. Thus, we perform a t-test with a confidence level of 95% regarding each hypothesis, which will allow the confident approval or rejection of the respective hypothesis.

6.7.2. Study Conduction

In conducting the controlled user study, we strictly follow the experiment design. We were able to recruit 16 participants from different areas and with varying levels of experience. In total, we recruited nine beginners, five advanced users, and two experts. We split the 16 participants into two groups of eight people each and tried to balance the groups as best as possible. After that, we trained the participants. Due to time conflicts, we were not able to train all participants at once and had to conduct several sessions.

We conducted the actual experiment with scenario A and B in a separate session, where we invited the participants individually. The individual

sessions gave us the chance to monitor the participants better and to measure personal values more accurately.

In the next section, we will elaborate on the results.

6.7.3. Study Results & Reporting

In the following, we will briefly report on the result of the study and only give relevant information. However, we have made all raw data publicly available[7].

After conducting the study, we were confronted with a set of measurements. First, we will look at the measurements we took during the study. For this, Table 6.4 summarises the result. The table shows all participants (first column), the measurements we took for the task with the standard toolkit (second to fourth columns), and the measurements for the parallel AT catalogue (columns five to seven).

In the summary section at the bottom of the table, the following characteristics are immediately noticeable, even without a detailed analysis:

Uncompleted tasks: Using the standard toolkit, two participants were not able to fulfil the task. Neither participant was a beginner, and one was an expert. We interviewed both participants and learned that they had tried to find a scripted or semi-automatic solution, which was not possible in the given time frame.

Performance increase: Comparing the mean completion time of the standard toolkit and the parallel AT catalogue shows that the parallel AT catalogue is on average more than three times faster.

Number of errors: The mean number of errors shows us two things. First, the participants make less than one error in average—in both scenarios. Even though the average number of errors is lower when using the parallel AT catalogue, we would have assumed a much higher error rate for the standard toolkit. This may indicate that we could have used a more complex scenario. The second observation is that the mean error rate is only slightly lower when using the parallel AT

[7]Raw Data: `https://doi.org/10.5281/zenodo.3755339`

	Standard toolkit			parallell AT catalogue		
Participants	**Completion time (s)**	**# of errors**	**Time in errors (s)**	**Completion time (s)**	**# of errors**	**Time in errors (s)**
Participant 1 ◇	1582	0	0	339	0	0
Participant 2 △	1697	2	43	433	0	0
Participant 3 □	1372	1	16	343	1	10
Participant 4 △	1255	1	14	324	1	11
Participant 5 △	1447	0	0	425	0	0
Participant 6 △	1058	0	0	268	0	0
Participant 7 △	1344	1	52	327	0	0
Participant 8 □	not finished	0	0	455	0	0
Participant 9 ◇	not finished	0	0	240	0	0
Participant 10 □	1269	0	0	504	2	39
Participant 11 □	1172	1	5	505	3	15
Participant 12 □	1417	2	25	566	3	45
Participant 13 △	1472	3	97	504	0	0
Participant 14 △	1411	0	0	349	0	0
Participant 15 △	1680	1	32	493	1	12
Participant 16 △	1577	0	0	548	0	0
Sum	19753	12	284	6623	11	132
Mean	**1410,93**	**0,75**	**17,75**	**413,94**	**0,69**	**8,25**
Standard Dev.	185,69	0,93	27,14	102,01	1,08	14,25

User Backgrounds: △ - Beginner; □ - Advanced; ◇ - Expert

Table 6.4.: Summary of the User Study (cf. [Zah20])

catalogue. This indicates that the extension is not helping to reduce the number of errors.

Time spent in errors: However, when considering the mean time spent in errors, we can assume that the errors are easier to fix when using the parallel AT catalogue.

Next, we look at questions five to seven from the questionnaire. Figure 6.11 displays the results in a likert plot.

All three plots show a strong tendency toward the parallel AT catalogue.

We found that 74,5% of all participants rated their performance with the parallel AT catalogue as fast or better, while only 12% would say the same of the standard toolkit. At the same time 69% rate their performance as equally slow when using the standard toolkit.

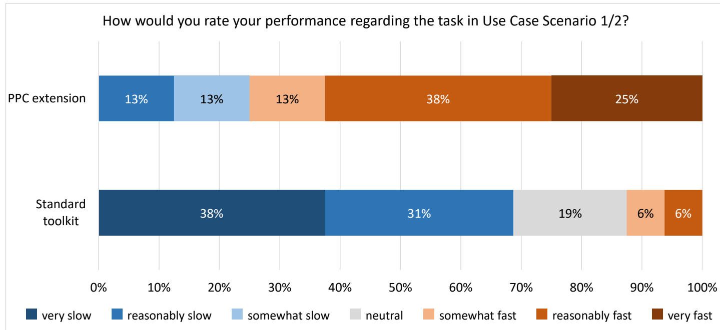
Additionally, 81% of the participants rate the amount of work required to fulfil the task as “little” when using the extension. None says it is too much. In contrast to that, all participants agree that the amount of work with the standard toolkit is much (19%) or too much (81%).

Finally, 94% of the participants rate the usability of the parallel AT catalogue as good and only 6% rate it as somewhat bad. In contrast to these numbers, the majority of the participants rate the usability of the standard toolkit as bad (13%) or very bad (69%) when it comes to parallel behaviour.

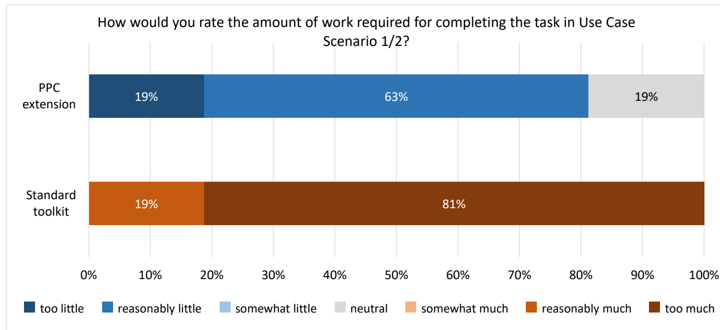
In addition to the evaluation by sight, we also performed a t-test evaluation for all of the goals, research questions, and corresponding hypotheses (see Figure 6.9), even though we are aware that the validity of t-tests is very limited, given the small sample size of 16 participants. To perform the t-test we followed the definition given by [WRH+12], formulated all H_0 hypotheses, and used a confidence interval of 95% in combination with a one-sided distribution table[8].

After performing the t-test, we can reject the H_0 hypotheses for goal I (improved usability measured by the questionnaire) and goal II (increased efficiency measured by the time needed). Thus, we have significant proof that the parallel AT catalogue increases the usability of Palladio when it comes to

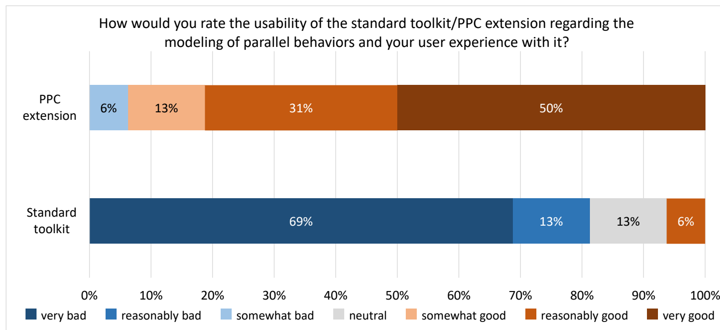
[8] `http://math.mit.edu/~vebrunel/Additional%20lecture%20notes/t%20(Student’s)%20table.pdf`



(a) Likert Plot of the Results from Question Five (rounded to integers)



(b) Likert Plot of the Results from Question Six (rounded to integers)



(c) Likert Plot of the Results from Question Seven (rounded to integers)

Figure 6.11.: Liker Plots of Questions Five to Seven [Zah20]

modelling parallel behaviour and increases the efficiency (reduces the time needed) of SA in creating models that include parallel behaviour. Regarding goal III (make Palladio less error-prone) and goal IV (reduce time in errors), we were not able to reject the H_0 hypotheses.

6.8. Transferability and Limitations

6.8.1. Transferability of the Parallel AT Catalogue

The parallel architectural template catalogue provides a set of the most common parallel patterns. It enables software architects to use parallel constructs in their software models quickly, easily, and efficiently.

Even though we focus on model-based performance prediction and therefore on languages like the PCM, we think that the approach is highly transferable. The PCM uses a UML-like syntax and semantics. Further, the AT method uses UML profiles to include the languages extension. Thus, transferring the approach to pure UML or to any other UML-like languages is easily doable.

Additionally, we did not do any domain specific pattern selection. Therefore, all of the identified, characterised, and realised patterns are of high value not only for software performance prediction, but for all computer science.

On the down side, we have to say that we included performance-specific attributes, like the overhead function modelling, in our patterns. These domain-specific characteristics are a valuable contribution to software performance engineers; however, they might not be of high relevance for other domains.

6.8.2. Limitations of the Parallel AT Catalogue

Even though the parallel AT catalogue, the parallel pattern taxonomy, and the formal semantics for parallel behaviour are of great benefit to software architects, we need to consider the limitations of this approach as well.

Parallel AT Process: In Section 6.3 we introduced and evaluated a process to include parallel patterns into modelling languages for performance predictions. Further, we were able to enhance the modelling language PCM to include specific characteristics of parallel behaviour. To develop this process we used the running use case example (matrix multiplication) and the state-of-the-art domain specific language (PCM). The underlying paradigm of the example is thread-based parallelisation and the PCM uses a UML-like syntax. Therefore, when using another paradigm (like message-passing) or another domain specific language (which is not UML-based), the process needs to be re-evaluated.

Pattern Catalogue: In the pattern catalogue we only included patterns that can be represented in a thread-based parallelisation paradigm. Pattern like AKKA Actors, which use a message-passing paradigm, were not included, since they will break the markovian properties of the underlying simulations.

Further, we did not include high-level parallelisation approaches, which are above the SEFF (software behaviour), and we explicitly excluded parallel components (e.g., in parallel executed container, services, etc.).

Visualisation: The approach is intended to support SAs. Therefore, we focused on a graphical language. Even though the PCM can be converted into any textual representation through model-to-model transformation, the design decisions we made might not hold true for a textual representation.

Evaluation: Even though the results of the empirical evaluation of the parallel AT catalogue favour the approach, the small sample size is an issue and can offer only weak statistical proof.

6.9. Summary of CB_1

In this chapter, we described the contributions we made with respect to the requirement $R_{modelling}$. To do so, we first identified the research need. Second, we showed that the current process of modelling parallel behaviour

for performance prediction with the help of state-of-the-art performance prediction tools (e.g., Palladio) is not only error-prone but also time-consuming. In addition to that, the predictions are innacurate as well.

In the next step, we formulated the research goal: To support software architects with an efficient way to express parallel behaviour in software models along with the necessary characteristics. Next, we created a method to enhance current modelling languages to include parallel patterns with the help of the architectural template method [Leh18]. While creating the method, we carefully evaluated different diagrams, view types, and enhancement concepts. To make a proof of concept, we used our running example (the matrix multiplication) and created the first parallel AT for the PCM. The evaluation of the working example verified the approach, and we ware able to:

1. increase the prediction accuracy by using an overhead function,
2. increase the efficiency through automatisation (use of AT), and
3. keep the function support of simulators and solvers.

Testing the approach encouraged us to continue building a full parallel architectural template catalogue. To do so, we performed a structured literature search to find 35 parallel patterns. We extracted the core characteristics of these patterns and created a taxonomy with five root patterns (see Figure 6.6). Out of this we successfully created a parallel AT catalogue which supports 4 out of 5 root patterns.

Finally, we conducted a controlled user study, in which we were able to empirically and significantly confirm that the parallel AT catalogue increases the efficiency and usability of the Palladio approach to modelling parallel software behaviour.

To wrap up, we can answer our research question as follows:

$RQ_{1.1}$: **Are software architects able to model even simple parallel concepts of highly parallel systems in an efficient way? Thereby, SA needs to focus on abstract performance relevant attributes on architectural level during early design time.**

Answer: *In an empirical user study using a controlled experiment, we were able to show that current state-of-the-art tools do not support SA in an efficient way.*

$RQ_{1.2}$: **Are software architects able to model the parallel software behaviour of an application with the help of current modelling languages, so that (a) the relevant performance characteristics are captured and expressed, and (b) all necessary information for performance evaluation is covered?**

Answer: *SA are currently not able to model (a) all relevant characteristics of parallel software, which results in (b) inaccurate performance predictions for parallel software in multicore environments.*

$RQ_{1.3}$: **How can software architects be supported in the task of creating accurate performance prediction models efficiently?**

Answer: *With the help of a parallel AT catalogue SAs can be supported in creating performance prediction models more quickly and with a higher user acceptance (usability). Furthermore, they can use the concept of overhead modelling to increase the accuracy of the predictions.*

$RQ_{3.1}$: **Are current simulation-based performance prediction approaches capable of predicting the performance of parallel and highly parallel systems accurately?**

Answer: *The experiments we performed in [FH16; FSH17] show that current state-of-the-art performance prediction approaches are up to 80% off when trying to predict the response-time for parallel applications in multicore environments*

With the parallel AT catalogue presented in this chapter, we make a significant contribution for SAs who want to make more accurate performance predictions for parallel software more quickly. The contribution also resolves and fulfils requirement $R_{modelling}$.

7. CB_2: Performance Curves for Parallel Behaviour

In this chapter, we will continue the research from contribution CB_1 (see Chapter 6) and still focus on $R_{accuracy}$ and $R_{modelling}$.

In CB_1, we presented a pattern catalogue extension for Palladio, providing the most relevant parallel patterns. We included a concept in the modelling process, which allows the SA to model the overhead and speedup behaviour with the help of performance curves. The biggest challenge here is to specify the overhead model, since this task requires a lot of experience and additional knowledge of the software and hardware.

Therefore, in this chapter, we investigate parallel performance-influencing factors (PPiFs), set up experiment-based performance evaluation, and extract performance curves for parallel application.

The overall goal is to extract and cluster characteristic performance curves, which can be provided to the SA. By the help of the performance curves, we want to enable SAs to easily define overhead functions and thereby further increase the performance prediction accuracy.

Figure 7.1 shows the structure and the research method followed in this section.

First of all, we are going to define the problem space, followed by the definition of the research goals and evaluation criteria. Next, we will investigate PPiFs, which we will use in the next steps to design the experiment setup. We will analyse the results from the experiment executions to extract performance curves, which we will integrate into Palladio. Finally, we will evaluate the approach using SPEC benchmarks.

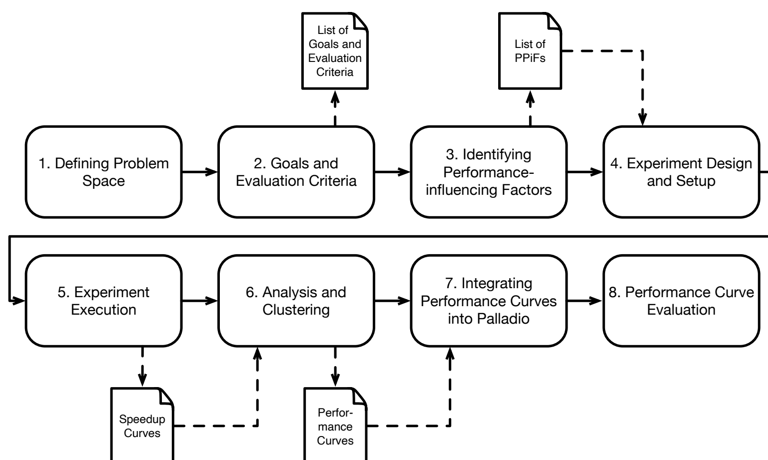


Figure 7.1.: Overview of the Research Method for Contribution CB_2

As a result of this contribution, we present (1) 14 lessons learned from the experiments and (2) deliver twelve performance curves to the SA. The performance curves represent the six most relevant software behaviours and increase the predictive power of Palladio. Thereby, we are able to increase the prediction accuracy up to 72% for the benchmark applu311.

Please note that significant parts of the work from steps one to three have been reviewed and published in [FBKK19]. In addition, the remaining steps are currently under review in [FSK+20].

Further, all results, raw data, and implementation details have been made available online:

Section 7.3 Load Test Generator Based on ProtoCom:
`https://doi.org/10.5281/zenodo.3828432`

Section 7.4 Experiment Raw Data:
`https://doi.org/10.5281/zenodo.3855492`

Section 7.5 Performance Curves:
`https://github.com/PalladioSimulator/Palladio-Addons-ParallelPerformanceCatalogue`

Section 7.7 Performance Curve Evaluation:
`https://doi.org/10.5281/zenodo.4081091`

7.1. Problem Space

As we have learned so far, the performance of parallel applications relies on a complex set of factors. Often these factors are interconnected and therefore, it is a tricky task to tell how PPiFs will affect the overall performance of an application without executing and measuring it. But even given the measurements, it is still a challenging and time-consuming task to determine the effect of each Parallel Performance-influencing Factor (PPiF).

In Chapter 6, we proposed an abstract approach to include speedup behaviour of parallel applications with the help of performance curves in the performance prediction models, by defining an overhead function. At the same time, we realised that defining these performance curves is a time-consuming and challenging task, which needs experience and additional knowledge of the software and hardware.

7.1.1. Idea

To save the SA the effort of specifying the overhead function, we want to provide the SA performance curves, which contain relevant PPiFs.

Figure 7.2 shows an example of a speedup curve based on the PPiFs' worker threads and resource demand type (see Chap. 5 for detailed information on the resource demands).

The diagram contains five different examples with an individual speedup behaviour characteristic for each case. This example can be mapped one-to-one to a two-dimensional performance curve.

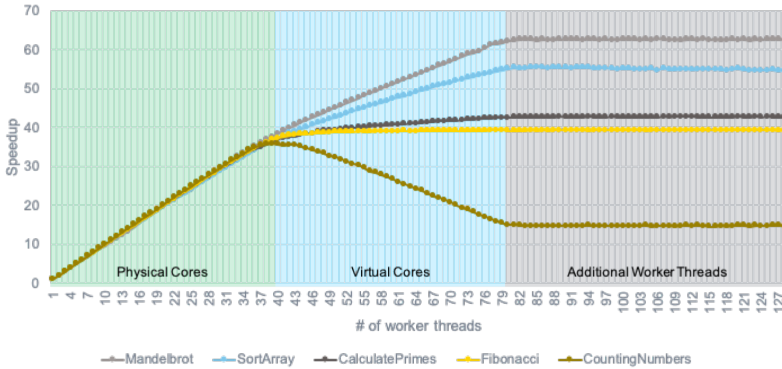


Figure 7.2.: Measurements of Speedup Functions for Different Resource Demands on a 40-Core System with Enabled Hyper-threading

Our idea is to integrate such performance curves into Palladio. That way the SA only needs to specify, e.g., the thread number and the resource demand type. The solver takes the performance curves into account and calculates the speedup behaviour based on the reference curve. In Section 5.1.7 we discussed a set of algorithms which can be exemplified to a resource demand type. We will use these alogrithms in the course of the chapter to investigate the resource demand types.

7.1.2. Problem Specification

Having a closer look at the topic, it becomes clear that defining performance curves is no straightforward task, and we have to overcome a set of challenges:

C_1 **Interdependent:** Often PPiFs are interconnected and it is difficult to isolate a single performance-influencing factor for evaluation, e.g., cache, memory bandwidth, and memory. Researching individual PPiFs and making the right deduction is a challenge.

C_2 **Variants of Behaviours:** The speedup behaviour can strongly vary and depend on the demand. As displayed in Figure 7.2 e.g., `MandelSet` con-

tinues to increase performance while the speedup of `CountingNumbers` decreases after a while. Identifying and clustering adequate types is a challenge.

C_3 **Types of PPiFs:** The variety of PPiFs ranges from fixed hardware-specific influencing factors, such as L1, to flexible software specific influencing factors, like thread pool size. Finding and selecting the right set of PPiFs is a major challenge.

Given these challenges we derive the following goals:

G_1 **Relevant PPiFs:** First of all, we want to determine the most relevant PPiFs.

G_2 **Complete Set:** Second, we need to provide a complete set of performance curves, either multiple ones or a single multi-dimensional one. The aim here is to have a performance curve for each specific demand—e.g., Mandel Set.

G_3 **Behaviour Matching:** For each specific demand, we need a performance curve that matches the behaviour as accurately as possible. Thereby, we do not aim for 100% accuracy, since the actual behaviour can vary for each implementation. We consider predictions that differ no more than 20% to be perfect, and a variation of 40% to be acceptable, as this value already greatly benefits the overall accuracy of parallel performance predictions.

Given these goals, we can derive two metrics to evaluate the final performance curves:

E_1 **Fitting:** How close is the performance curve to the actual behaviour?We can get this value by comparing the performance curves to measurements form the executions.

E_2 **Completeness:** How many specific demands can we cover with our set of provided performance curves? To evaluate E_2, we plan to use benchmark sets. The more benchmarks we can cover, the better.

Taking the challenges, goals, and evaluation metrics into account, we can define the research method next.

7.1.3. Research Method

To estimate the performance curves, we combine the approach of experiment-based performance model derivation proposed by [Hap08] and the process of extracting performance curves by [WHW12]. This process is displayed in Figure 7.1. Concretely, we first want to determine a set of relevant PPiFs by scanning the literature and conducting expert interviews. Next, we rank the PPiFs and start to build a performance curve for the most relevant ones. If we are satisfied, we continue; if not, we consider additional PPiFs.

For each PPiF, we set up an experimental design to monitor and measure the behaviour of the software performance. In our case, we focus only on the execution time, specifically, the speedup behaviour. From the measurements, we perform statistical analysis and clustering to determine a set of the relevant performance curves. Finally, we integrate them into Palladio, utilising overhead functions, and evaluate their accuracy.

The research method, along with the collection of the PPiFs, was published, reviewed, and accepted in [FBKK19]. Besides that, major portions of the measurements were gained in collaboration with student projects [Gre19].

7.2. Parallel Performance-influencing Factors

The first step towards performance curves is to identify a list of potential PPiFs. To do so, we perform a literature review and interview experts from different domains, like SPE, HPC, and operating system domain. Next, we prioritise the PPiFs based on the results from the expert interviews.

In the following, we first present the outcome of the PPiFs-collection and the interviews. Afterwards, we rank the list based on the insights we gained during the discussions.

7.2.1. PPiFs Collection

The following list of PPiFs represents the outcome of a literature review [Söh18] and expert interviews we performed. For the latter, we interviewed

four software performance experts within our department, seven HPC experts from the University of Dresden, Hasso-Plattner Institute, and Karlsruhe Institute of Technology (KIT), and three experts on parallel execution in embedded systems from the University of Chemnitz.

The following list is quoted verbatim from [FBKK19]; it is categorised into two groups (configurable and fixed PPiFs) and contains the subset of all PPiFs that the experts agreed on:"

7.2.1.1. Configurable PPiFs

Configurable factors are properties which can be directly configured or influenced by the software developer and therefore adjusted to the given hardware or scenario. Often auto-tuners are used to find the best configuration for these properties on a given system.

Parallelisation Strategy: The parallelisation strategy describes the parallelisation paradigm or pattern used, e.g., Java Threads with a master-worker pattern, OpenMP, or ACTORS.

Thread Pool Size: The thread pool size specifies the number of worker threads. Typically, software threads are mapped to worker threads and then to hardware threads. Only worker threads are active.

Number of Threads: This is the number of total spawned threads in the application. In other words, in a Java application spawning, a thread for each task executed in parallel is possible. By using a thread pool, these threads are scheduled.

Software Caches: Software caches can influence the performance of the software significantly.

Data Locality: Usually, data is stored in the memory belonging to the core which first touches/creates the data. So this core has the optimal latency to access the data while other cores have significantly higher latency.

7.2.1.2. Fixed PPiFs

In contrast to configurable PPiFs, fixed PPiFs are given by the considered application or the infrastructure used, and cannot be influenced by the software developer.

Type of Resource Demand: The type of resource demand is given by the kind of task performed on the CPU, i.e., processor-intensive tasks (like calculating Fibonacci numbers) or I/O-intensive tasks (like sorting an array).

Memory Design: Memory design is a hardware-specific characteristic and defines the layout of CPUs, caches, and main memory. It also describes how these components are interconnected.

Memory Bandwidth: Memory bandwidth specifies the characteristics of the interconnections of the memory design, i.e., how many lanes are available, what is the total throughput, and how many components share the connection." [FBKK19]

We do not claim this list to be complete, but it does contain the relevant factors for parallel execution that we located in literature, and abtained from the expert interviews.

7.2.2. Prioritising

Now that we have the list of PPiFs at hand, we need to prioritise the list. The prioritisation is essential to decide which factor to take into account first. Considering all factors at once increases the effort significantly and makes both the extraction of performance curves as well as the decision for the SA more complex.

So we not only take into consideration the effect of the factors, but also the challenge for the SA to retrieve this information. Table 7.1 shows the prioritised list worked out with the expert board.

Highest ranked are the threads and the thread pool size. It seems logical that these two factors influence performance the most and directly. We could also add the number of hardware cores here, but we included that in the

Prio.	PPiF	Prio.	PPiF
1.	Number of Threads	5.	Memory Design
2.	Thread Pool Size	6.	Memory Bandwidth
3.	Type of Resource Demand	7.	Software Caches
4.	Parallelisation Strategy	8.	Data Locality

Table 7.1.: Prioritised list of PPiFs after ranking by experts

thread pool size. If there is no multicore hardware available, considering threads would not make sense. Even though context switches are a relevant factor as well, this topic is already covered by J. Happe [Hap08].

Next, we rank the type of resource demand, because the board agreed upon the fact that the kind of operation has a direct impact on the parallelisability of the problem, and therefore on its speedup. In contrast to that, the decision regarding the parallelisation strategy is not as clear. The board agreed that the paradigm used to parallelise an application affects performance. But the committee could not decide on the level of impact. The main argument against a high ranking was that, correctly implemented, all paradigms result in a good speedup behaviour.

For factors five to eight, the board again agreed on their impact, especially that data locality and caches have a high impact on the speedup behaviour. However, we rank data locality low, because it is hard for the SA to consider that in architectural models. Further, we ignored software caches for now.

7.3. Experiment-Based Performance Evaluation

In this section, we describe the experimental design and setup, the hardware environments, the experiment results, and the extraction of the performance curves.

7.3.1. Experiment Design

The outline of the experiment is sketched in Figure 7.3. Thereby the first step is to get and generate typical resource demands. For this purpose we use ProtoCom[1] (see Section 2.4.2.1).

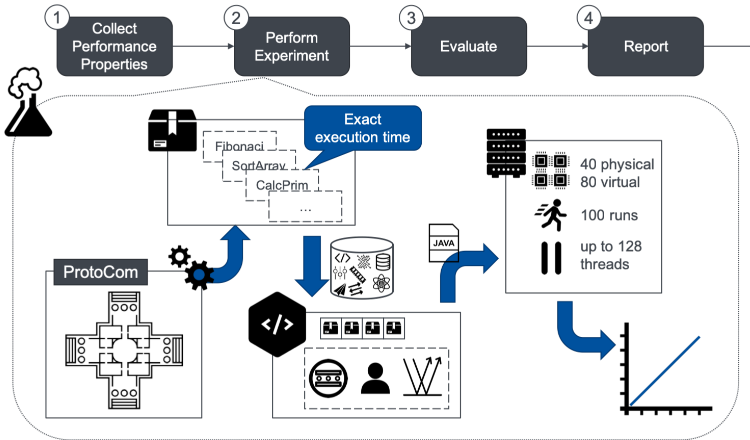


Figure 7.3.: Overview of Experiment Setup using ProtoCom as Resource Demand Factory [FBKK19]

ProtoCom: ProtoCom provides five different types of basic resource demands: Mandel set, sorting arrays, counting numbers, calculating primes, and calculating Fibonacci numbers. In addition to that, we implemented one additional demand—multiply matrices—and adjusted other demands, like sorting array, to be able to specify the array size. All implementations of the resource demands are given in Appendix A.2.

ProtoCom enables us now to generate work packages of the six specific primitive resource demands. The advantage of using ProtoCom is that we can specify the exact runtime (i.e., five seconds) of these packages in a given environment [BDH08]. We use this characteristic to generate several independent work packages of the same resource demand, which have zero

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

interdependencies. Thus, we can guarantee a pure workload on the CPU without communication, waiting, or locking side effects.

Parallelisation: In the next step, we take these generated packages, add them to a queue, and build a parallelisation approach around it. In total, we support four parallel paradigms: Java Threads, Java Streams, OpenMP, and AKKA ACTORS.

Each paradigm can take the queue and execute it in parallel. Thereby, we can specify the thread pool size and can measure the pure execution time of the queue-execution step.

Finally, we can generate a runnable jar file, which can be executed with the desired parameter set on the target platform. The complete source code is available online[2].

Experiment Execution: In the last step, we take the runnable jar file and deploy it on the target platform. For each platform, we perform multiple runs, always changing only one parameter: Thread pool size or parallelisation paradigm. We run each configuration multiple times and vary the thread pool size from one to three times the number of physical cores available on that platform.

While performing each run, we measured not only the runtime but also the cache behaviour. Since measuring low-level metrics in this way is not supported by the JVM, we used PAPI API[3] and perf[4].

7.3.2. Experiment Environment

To investigate the behaviour of different hardware environments, we performed our experiment on multiple target platforms. The characteristics of all machines used are displayed in Table 7.2. We use three dedicated servers of different dimensions. The smallest has 12 physical cores and the largest 96 physical cores.

[2]Load Test Generator: `https://doi.org/10.5281/zenodo.3828432`

[3]`http://icl.cs.utk.edu/papi/`

[4]`https://perf.wiki.kernel.org/index.php/Main_Page`

	Hardware Enviorments			
Attribute	**Server Stuttgart**	**Potsdam Small**	**Potsdam Large**	**BwUniCluster**
Type	Dedicated	Dedicated	Dedicated	Cluster
Location	Stuttgart	HPI Potsdam	HPI Pottsdam	Karlsruhe
OS	Ubuntu 18.04.2 LTS	Ubuntu 16.04.6 LTS	Ubuntu 16.04.6 LTS	Red Hat Enterprise Linux 7.7
JDK	OpenJDK 11.0.4	OpenJDK 11.0.4	OpenJDK 11.0.4	OpenJDK 13
# Cores	96	12	40	14 per Node
Hyperthreading	enabled	enabled	enabled	enabled
CPUs	Intel(R) Xeon(R) Platinum 8168 CPU	Intel(R) Xeon(R) CPU E5-2640	Intel(R) Xeon(R) CPU E7- 4870	Intel(R) Xeon(R) CPU E5-2660 v4
Clock rate	2.70 GHz	2.5 GHz	2.40 GHz	2.0 GHz
# CPUs	4	2	4	2 Nodes
L3	33 MB*	15 MB*	30 MB*	35 MB*
L2	1 MB**	256 KB**	256 KB**	256 KB**
L1	32 KB**	32 KB**	32 KB**	32 KB**
RAM	376 GB	32 GB	896 GB	2x 128GB

*shared cache per processor **private cache per core

Table 7.2.: Overview of the hardware environments and their configuration

7.4. Measurements and Results

We execute the above-described experiment setting for all 96 variations. The variations include the six resource demands, the four parallelisation paradigms, and the four different hardware settings. Thereby, we measure the execution time as well as the L2, L3, and main memory access where possible[5].

We execute all experiments for all the demands with a package execution time of $0.2s$. Thereby, we configure the total amount of packages for each hardware individually, always three times the number of available cores. Using the same number of packages for all the four hardware settings would mean having to pick the highest value. This would result in very long execution times on smaller hardware environments.

In total, we end up with over 70,000 measurements in over 800 experiment runs. Due to this extensive amount of data, we are not able to show and discuss all the results in detail. In this section, we present the results for the server in Stuttgart only, which are exemplary. The results for the hardware in Potsdam and the multi-node cluster (cloudbw) are attached in Appendix A.4. Even though we only show the results from Stuttgart here, we discuss noteworthy results of all the experiments.

A full description of the experiment setup, execution, and discussion is available in the supervised student thesis [Gre19]. Further, all results and raw data are publicly available online[6].

7.4.1. Result Report Server Stuttgart

For the sake of understanding, we first separate the performance/speedup and the memory behaviour aspect. Thus, we first report the performance of the individual experiment runs concerning the thread pool size. Later, we have a closer look at memory behaviour, and finally, we bring both aspects together.

[5]Not all hardware supports reading the performance counter for L1, L2, and L3 cache
[6]Experiment results raw data: `https://doi.org/10.5281/zenodo.3855492`

7.4.1.1. Performance Behaviour

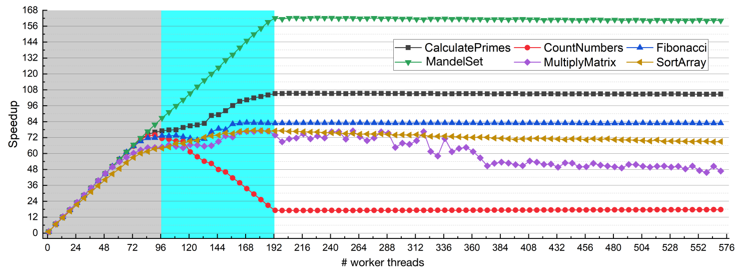
Figure 7.4 shows the measurements using a speedup chart for the different parallelisation paradigms and resource demands. The x-axis indicates the number of used worker threads. This number represents the number of active threads (i.e., the thread pool size). While each worker thread is directly mapped to a processing unit, the threads in the system are assigned using the thread pool to worker threads.

The y-axis displays the speedup. We calculate this value based on the execution time of a single thread application (i.e., by using only one worker thread). To increase the readability of the diagrams, only every sixth data point is displayed. The line between the data points represents the skipped values.

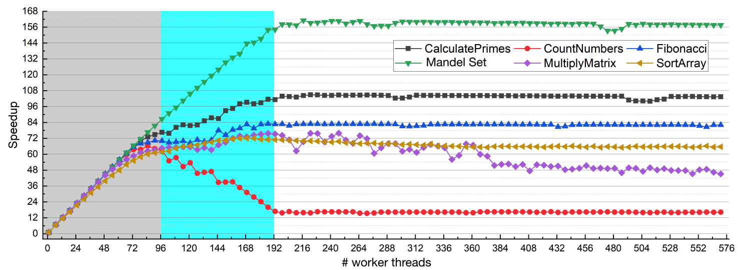
The first area from the left (from 0 to 96 worker threads) indicates the field where each worker thread can be mapped to a physical core. The second area from the left (from 97 to 192) shows the field where, due to hyper-threading, each worker thread can be mapped to a virtual core. The third area from the left (from 193 to 576) represents the area where we increased the number of worker threads even further. In this area, not all worker threads can be directly mapped to cores, which means that the scheduler either has to switch tasks, and therefore handle context switches, or suspend worker threads until a core is free.

At this point, we notice three characteristics:

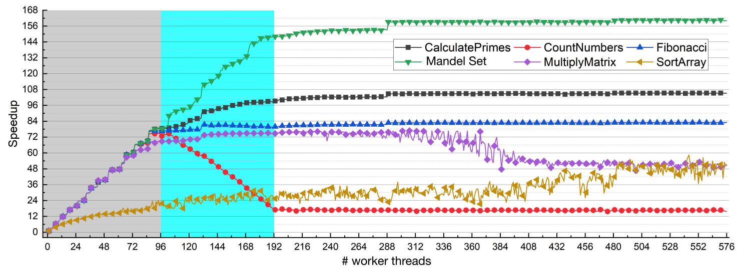
1. The speedup behaviour of AKKA Actors differs a lot from the behaviour of the other paradigms. The root cause of this can either be an implementation error, or a characteristic of the framework. Since we double-checked the implementation multiple times in code reviews, we assume the root cause to be in the AKKA Actors framework. Due to this fact, we will not consider the results for the AKKA Actor framework in the following.
2. For each of the three areas, we see different behaviours for all demands. While in the first area (0 to 96 worker threads) the speedup is close to a linear behaviour for all of the demands, there is a spread of the speedup in the second area (97 to 192 worker threads). On the one hand, I/O-intensive tasks like Mandel Set (lots of small read and



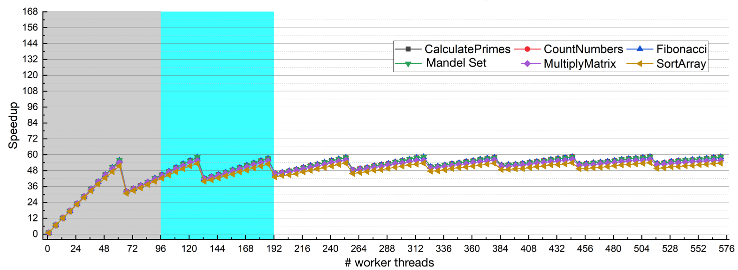
(a) Speedup Curve for all Demands Using Java Threads



(b) Speedup Curve for all Demands Using Pyjama (OpenMP)



(c) Speedup Curve for all Demands Using Java Streams



(d) Speedup Curve for all Demands Using AKKA Actors

Figure 7.4.: Speedup for Different Parallelisation Paradigms [Gre19]

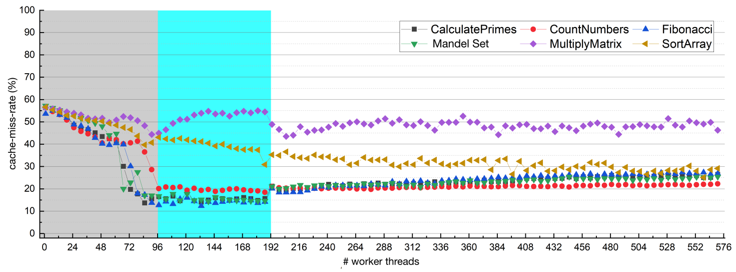
write operations) can benefit from hyper-threading and continue to speed up. Even though the speedup is not as great as before, it is still linear. On the other hand, a processor-intensive task, like calculating primes or Fibonacci numbers, cannot benefit much from hyper-threading and stays constant. Further, very I/O-intensive tasks, like sorting arrays or calculating matrices, show a rather bad performance in area two, compared to hardware environments with smaller core numbers. A hypothesis here is that due to cold caches and unfortunate memory architectures, the hyper-threading effect is abrogated. Noteworthy is the decreasing performance of the counting numbers demand as well. In the third area (from 193), we can see a performance stagnation with low tendency to a performance degression.

3. Ignoring AKKA Actors, the speedup behaviour of the individual demands does not differ much for the paradigms. For example, the speedup curve for the Mandel Set demand is similar for Java threads, Java streams, and pyjamas. Thus, we can say that the paradigm used does not have a great impact on the speedup behaviour. An outlier here is the sorting arrays demand, but only for Java streams.

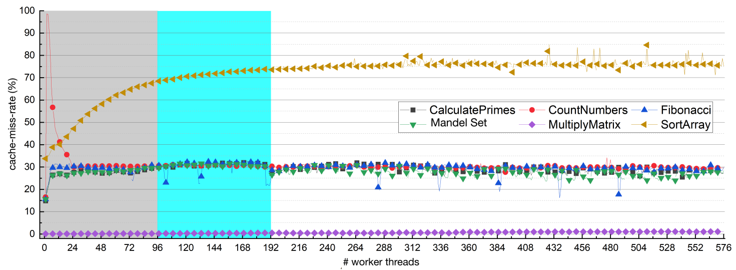
7.4.1.2. Memory Behaviour

Besides the performance of the parallelisation paradigms and resource demands, we also measure memory behaviour. Here we measure the L2 and L3 cache miss ratio and the total number of cache accesses. Again, due to the extensive amount of data, we focus in the following only on the measurements taken from the dedicated server in Stuttgart, the parallelisation paradigm Java threads, and limit the scope to the L2 and L3 cache accesses and miss rate.

Figure 7.5a and 7.5b show the cache miss ratio for the L2 and L3 caches. Thereby, the x-axis shows the number of used worker threads again. The y-axis shows the percentage of cache misses (a lower number is better). In addition to that, Figure 7.6a and 7.6b show the total number of cache accesses for L2 and L3 on the y-axis.

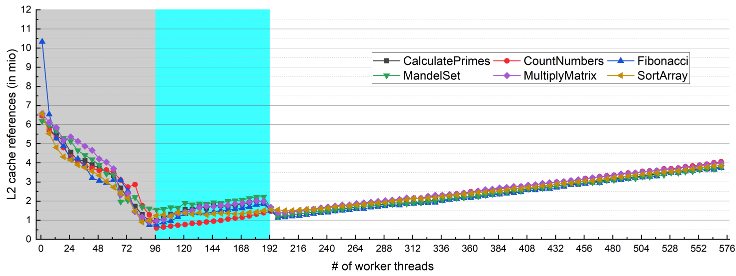


(a) L2 Cache Hit/Miss Rate for all Demands Using Java Threads

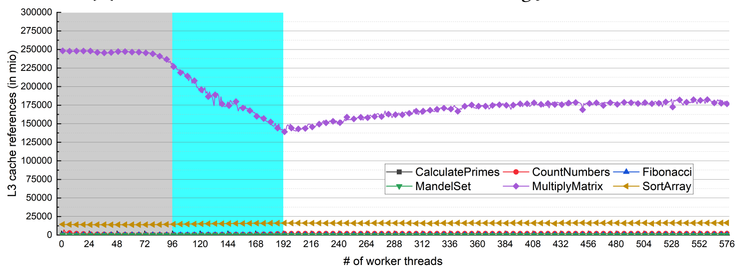


(b) L3 Cache Hit/Miss Rate for all Demands Using Java Threads

Figure 7.5.: Cache Miss Rate for Java Threads on the Server in Stuttgart [Gre19]



(a) Total L2 Cache Accesses for all Demands Using Java Threads



(b) Total L3 Cache Accesses for all Demands Using Java Thread

Figure 7.6.: Cache Accesses for Java Threads on the Server in Stuttgart [Gre19]

The measurements give us detailed insights on memory behaviour. We highlight the following characteristics:

1. On all machines for all resource demands, but only for Java threads, we can observe a high cache miss rate on L2 cache for a low number of worker threads. As indicated in Figure 7.5a, the optimal cache miss ratio is achieved when all cores are utilised. One reason for this is that the L2 cache is core specific and not shared. Therefore, the more cores we can utilise, the more cache we have available. Thus, the total amount of cache size increases. Again the sorting array demand is an outlier for this observation.

2. All other parallelisation paradigms have rather constant cache miss rates on L2.

3. Considering the L3 cache, the sorting array demand has more cache misses, when increasing the worker threads.

4. The matrix multiply demand fits completely in the cache and needs no main memory accesses.
5. As shown in Figure 7.6a and 7.6b the Mandel Set, Fibonacci and calculating primes demands have very few L3 cache accesses. Therefore, we can assume that all data for these demands fit in L1 and L2. This effect is even more visible on smaller hardware.
6. Multiply matrix demand has significantly more L3 cache references.

7.4.2. Comparison of Parallelisation Paradigms

In the next step, we compare the performance for all hardware, resource demands and parallelisation paradigms. For this, we focus on each resource demand type and compare the performance of the parallelisation paradigms. We make the comparison for each hardware separately.

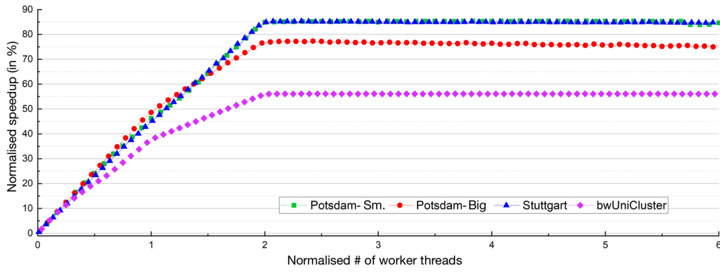
We notice that for each resource demand, the speedup behaviour is similar—no matter which parallelisation paradigm we use. Here we have to note the unexplainable behaviour of the AKKA Actor implementation again, which we neglect in the comparison.

On the one hand, this is surprising, because we assumed that the parallelisation paradigm has an impact on the speedup behaviour. On the other hand, we do not compare the absolute performance. That means that the parallelisation paradigm can have an impact on the absolute performance, but scales similarly.

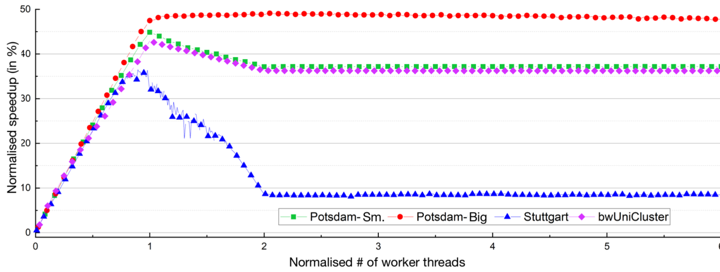
Further, we notice that we achieved a very good overall speedup. This is because we used the packages from Protocom, which are independent and place the parallelisation paradigm on top.

7.4.3. Comparison Server

Next, we are interested in how the hardware setting influences the speedup behaviour. To analyse it, we first need to normalise the results for all machines. Normalising means we divide the number of worker threads by the number of available physical cores. Further, we describe the speedup as a



(a) Comparison of the Four Hardware Environments Using Java Threads and the Mandel Set Demand



(b) Comparison of the Four Hardware Environments Using Java Threads and the Count Number Demand

Figure 7.7.: Comparison of the Four Hardware Environments Exemplified by Using Java Threads, Mandel Set, and Count Number Demand

relative value in percent. In theory, a speedup of 100% is possible. As an example, if we take the large server in Stuttgart, which has 96 physical and, due to hyper-threading, 192 virtual cores, a speedup of 100% would mean utilising all virtual cores optimally and achieving an absolute speedup of 192. In Figure 7.7 we use the results from Mandel Set (best speedup) and the count number (worst speedup) demand to exemplify the results of the comparison while using Java threads.

First, we focus on Figure 7.7a, which shows the speedup behaviour for the four different hardware environments using the Mandel Set demand and Java threads. This demand performed the best in all the experiments, and shows the best parallelisation characteristics. As we can see, the server in

Stuttgart and the small server in Potsdam show the best and almost identical behaviour. The big server in Stuttgart shows a slightly better behaviour before the number of physical cores are hit (up to one). However, in area two (from one to two) it cannot benefit as much from hyper-threading. The multi-node server (bwCluster) shows the weakest performance. However, for all three areas, all machines show the same characteristics. Only the gradient of the charts differs.

Next, we focus on Figure 7.7b. This figure shows the speedup behaviour for the four different hardware environments using the count number demand and Java threads. This demand showed the worst speedup behaviour in all the experiments. Having a look at the diagram, we notice four peculiarities: First, the speedup in area one (zero to one) is almost alike for all environments. Second, on the hardware in Stuttgart, speedup already flattens at around 0.8 or 80 cores (see also Figure 7.4a). Third, three out of four show decreasing performance in area 2 (one to two). While the small hardware in Potsdam and the multi-node cluster show similar behaviour, the server in Stuttgart underperforms, and the big server in Potsdam shows no performance decreases at all. Fourth, in the third area, all hardware shows the same behaviour again.

In summary, we can state that there are slight differences when considering speedup behaviour among all the different hardware environments. However, the essential characteristic is mostly the same. This is an important observation, because it allows us to extract performance curves from our measurements regardless of the hardware used. Further, we will be able to generalise the performance curves for all kinds of general-purpose hardware environments using a similar architecture.

7.4.4. Lessons Learned

After we conducted the experiments and displayed the key results of the measurements, we can state interesting insights. These insights are not only relevant for our research question and the next step—extracting performance curves—but also show informative facts about parallel computing in general. In the following, we list all relevant aspects:

L_1: We never achieved a perfect speedup (speedup equals the total number of physical or virtual cores), which confirms our hypothesis $H_{2.3}$, that there are further performance-influencing factors in a system beyond the pure number of physical cores.

L_2: In area one, where all worker threads can directly map to the CPU cores, we see a similar behaviour for all demands and parallelisation paradigms on all devices, and close to linear speedup.

L_3: In area two, where all worker threads can directly map to virtual cores, we can see that only I/O-intensive tasks are able to benefit from hyper-threading and gain additional speedup. The processor-intensive tasks do not speed up any more. Very I/O-intensive tasks even lose performance due to hyper-threading. We assume a lot of context switches, cold caches, and busy memory buses to be the reason for this.

L_4: The AKKA Actor framework shows an inexplainable, strange, and bad parallelisation behaviour. The root cause for this, we assume, is in the implementation. Due to this fact, we neglect this parallelisation paradigm for further considerations.

L_5: Besides the AKKA Actors, all other parallelisation paradigms show a similar speedup behaviour. This observation was surprising for us, and contrary to hypothesis $H_{2.1}$. Therefore, we say that the parallelisation paradigm has no great impact on the performance.

L_6: Even though we were able to show that the hardware architecture has an impact on the overall speedup behaviour and therefore confirm hypothesis $H_{2.2}$, the impact of the hardware architecture was medium to low. Again, we did not consider the absolute performance, but the relative speedup behaviour.

L_7: The cache behaviour for each resource demand is characteristic for the number of worker threads—regardless of which paradigm or hardware is used. Both the hardware and the paradigm influence the intensity of the cache miss rate, but the overall characteristic remains. For example, the cash-miss rate for the sorting array demand increases for a higher number of worker threads.

L_8: The counting number demand shows exorbitantly bad speedup behaviour for all paradigms in all environments. The speedup behaviour becomes worse the more cores are utilised. Further, counting numbers contradict benefits from hyper-threading.

L_9: While most paradigms show a dynamic in the L2 and L3 cache miss behaviour, the Java stream demand surprises with a rather constant cache miss rate for L2 and L3 caches. Further, in comparison to other demands (e.g. Java threads), which show a better cache hit rate for a higher number of worker threads, there is no noticeable difference in the speedup behaviour of Java streams.

L_{10}: The Mandel Set, calculating primes, and also Fibonacci demand show a comparably low number of cache references. For example, on the hardware in Stuttgart using the Java threads paradigm and 196 worker threads, the sorting array demand has 910 times more L3 cache references than the Mandel Set demand. Demands with low cache references show a better speedup behaviour, especially during hyper-threading.

L_{11} Thus, demands with I/O intensive tasks like the Mandel Set demand, where all data fits into the core-specific cache (L1 and L2, can benefit from hyper-threading most and show the best speedup behaviour.

L_{12} I/O intensive tasks, like sorting array or multiply matrix demands, show a better relative speedup behaviour on smaller machines than on larger ones. We assume the reason for this is the limited memory bandwidth. On large machines where many cores can be utilised, the I/O demand is much higher than on smaller machines, e.g., if an application needs to read in total 10 Gbyte of data from the memory.

L_{13} The multi-node system (bwUniCluster) shows the worst speedup behaviour, in comparison to dedicated hardware.

L_{14} We were not able to observe any impact of the CPU frequencies, nor of the cache sizes on the speedup behaviour.

7.5. Extracting Performance Curves

In the course of this section, we describe the process of extracting performance curves from the measurements. Due to the massive amount of measurements, we follow a structured method to extract the data. This process consists of four steps: normalisation, clustering, staging, and extraction. We describe each step in detail in the following.

7.5.1. Normalisation

First of all, we decide to abstract from the actual measurements. To do so, we create speedup curves for each experiment run. As a reference for the speedup curve, we always use measures from the single-thread run. That way, we do not need to compare actual measurements with each other, but have a more abstract view on the data.

Next, we face the challenge of comparing measurements from different machines. Since each hardware environment has distinct characteristics and a different number of cores, the maximal possible speedup differs as well. To still be able to compare measurements from different machines, we need to normalise the data. As a normalisation factor, we used the number of cores available in each setting. As described in Section 7.4.3 we divide both the speedup and the number of worker threads by the number of available cores in the system. As a result, we get normalised values for all the machines, which we are able to compare. Figure 7.7 gives one example for the parallelisation paradigm: Java threads and the resource demand Mandel set. As depicted in the figure, the speedup of the machine in Stuttgart is almost the same as the small machine in Potsdam. For example, the x-axis value of 2 stands for the use of 192 worker threads in Stuttgart and 24 worker threads in Stuttgart. In both cases, this is twice as much as the number of physical cores. Both achieve a relative speedup of 85%, which is an absolute speedup of 160 in Stuttgart and 20 in Potsdam.

7.5.2. Clustering

After we are able to compare all measurements with each other, we have to perform clustering to get the curves which behave similarly. In our case, we perform a manual clustering based on the observations of the speedup curves. As shown in Figure 7.4, all demands have unique behaviour. Therefore, the first cluster criteria are the resource demand type. Next, we compare the speedup behaviour for the given hardware environment and parallelisation paradigm for each demand type. As stated in Lesson L_5, we can confirm that the choice of the parallelisation paradigm has no significant impact on the speedup behaviour. Thus, we do group by parallelisation paradigm. However, as stated in Lesson L_4, we assume a bug in the AKKA Actors framework caused the unnatural behaviour. Therefore, we neglect these measurements for further consideration.

A greater impact on the behaviour has the choice of hardware environment—as illustrated in Figure 7.7. For all but the counting number demand, the difference between the four environments lies in a corridor of maximum 30%. Thereby, the dedicated servers do behave similarly, and only the virtualised bwUniCluster behaves differently. Thus, we decide to separate virtualised and dedicated systems.

7.5.3. Staging

Besides clustering, we noticed that the speedup behaviour differs sharply when reaching specific numbers of worker threads. Therefore, we introduced three stages. The three stages align with the three areas in the previously shown diagrams. Stage one starts with one worker threads and goes up to the number of physical cores, the second stage goes from here to the number of virtual cores, and the third stage goes from here until infinity.

7.5.4. Extraction

In the final step, we extract the performance curves from each cluster and stage. To do so, we use linear regression. Thereby, we consider all speedup

	$f(x)$ for **Stage**		
Demand Type	**1**	**2**	**3**
CountNumbers	$0.438x$	$-0.171x + 0.572$	$-0.0038x + 0.230$
MatrixMultiplication	$0.412x$	$0.043x + 0.357$	$-0.0148x + 0.472$
FibonacciNumbers	$0.452x$	$0.026x + 0.417$	$0.00341x + 0.456$
PrimeNumbers	$0.449x$	$0.096x + 0.333$	$0.00140x + 0.536$
SortArray	$0.407x$	$0.151x + 0.252$	$-0.0129x + 0.573$
MandelSet	$0.458x$	$0.314x + 0, 206$	$0.00940x + 0.791$

Table 7.3.: Extracted Performance Curves for Dedicated Machines Based on the Speedup Behaviour of the Demands

curves in a cluster and stage, take the average, and extract a linear function using regression. For the first two stages, we gain very fitting curves (r-value above 0.90 for a confidence interval of 0.95). For the third stage, the variance of the measurements is higher. Thus, the resulting curves are not as fitting (r-values between 0.3 and 0.87). Table 7.3 shows the performance curves for dedicated machines for each demand type and stage (Appendix A.2 shows the performance curves for virtualised hardware). Additionally, Figure 7.8 visualises the performance curves. The x-value is the normalised value of the worker threads ($workerThreads/physicalCores$). The y-value gives the relative speedup concerning the maximal possible speedup ($speedup/virtualCores$).

7.5.5. Using Performance Curves: An Example

The SA can now use the above performance curves to correct the performance predictions—not only from Palladio, but from any performance prediction tool. To utilise the performance curves, the SA needs information about the available cores in the system, the number of worker threads, and the kind of resource demand. For example, assume we have a dedicated machine with 30 physical cores, using 45 worker threads, and have a resource demand-type which is close to the sorting array demand. First, we have to calculate the normalised x-value: $x = 45/30$, which is 1.5. After checking Table 7.3, we pick the following performance curve: $f(x) = 0.131x + 0.250$.

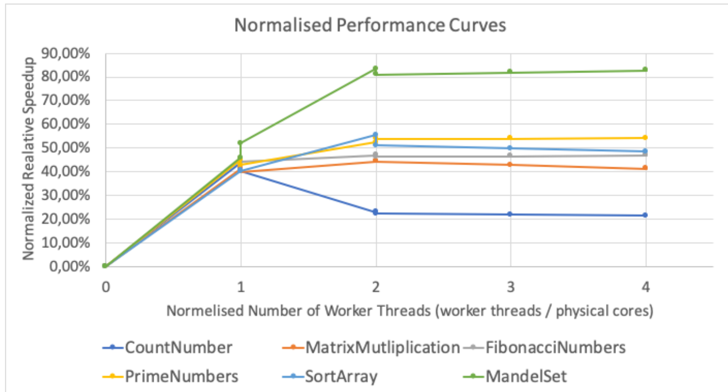


Figure 7.8.: Comparison of the Four Hardware Environments Using Java Threads and the Mandel Set Demand

Inserting the above values, we end up with: $f(1.5) = 0.131 * 1.5 + 0.250$, which is 0.45. This is the relative speedup calculated by the performance curve (absolute is 27).

In contrast, Palladio assumes a linear speedup which is in our example 45 (absolute) or 0.75 (relative). So we can now correct any performance prediction given from Palladio by the factor 0.6. For example, imagine our Palladio simulation takes $200s$. We multiply the Palladio result with the factor and end up with an output of $120s$.

Of course, this is a lot of manual effort. Therefore, in the next section we discuss integrating performance curves into Palladio for automated calculations.

7.6. Palladio Integration

To integrate the performance curves into Palladio, we need to alter the performance predictions. One way of doing so is to include the performance curves into the simulators. Another way, which we follow here, is to use the overhead concept introduced in CB_1.

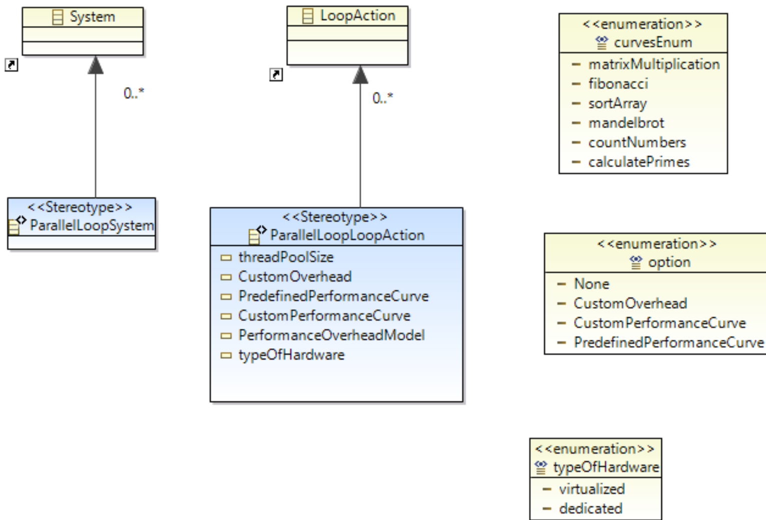


Figure 7.9.: Profile Example for Parallel For-loop

In short, we alter the parallel patterns to include the performance curves directly into the patterns. Further, we use the QVT-o transformations to automatically estimate the right overhead, add it to the model, and run the simulations.

In the following, we briefly discuss changes made to the profiles and the difference in the workflow for the SA. The full implementation details and the source code are available in the git repository of the parallel AT catalogue.

7.6.1. Profile Extension

To include the performance curves into the parallel ATs, we first need to alter the profiles of each AT. Figure 7.9 shows the final AT given the parallel loop AT.

We include three new enum types, enabling the SA to choose whether to use a custom overhead function, a custom performance curve, a pre-defined performance curve, or no overhead model at all. The first enum defines

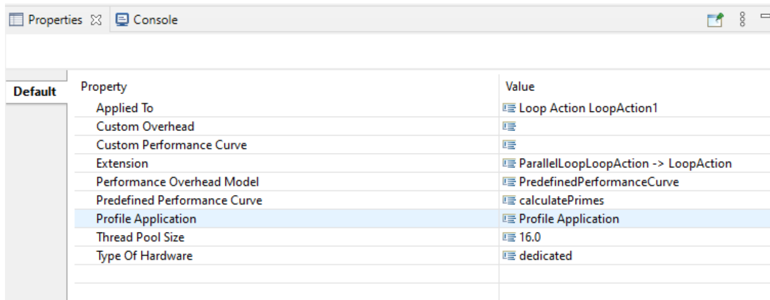


Figure 7.10.: Property View of the Applied Parallel Loop AT

whether to use a performance curve or not, the second enum specifies which demand-type curve to choose, and the third one whether to use the performance curves for virtual or dedicated hardware. Further, we add the required fields for a custom performance curve.

7.6.2. Workflow Adaptation

To use the performance curves, the SA first needs to model the software, hardware, and usage model as normal. Next, the SA needs to apply a parallel AT from the parallel pattern catalogue. Figure 7.10 shows the property view of the applied catalogue. Here, the SA can choose to use a performance curve and picks the desired curve for his resource demand and hardware type. If desired, he can also input his own performance curve.

After setting all properties, the SA can run the simulation using experiment automatisation. Within the QVT-o transformation, the properties are interpreted, the correct performance curve is picked, and the overhead is added.

7.6.3. QVT-o Transformation

Running the simulations with the AT method extension, will call the QVT-o script of the parallel AT and trigger the m2m transformation, before the actual simulation takes place.

We altered the QVT-o scripts to now automatically calculate the correct overhead by picking the right overhead function for the given configuration. The calculation of the overhead happens according to the example given above (see Section 7.5.5). We transform the time units in resource demand and add the resource demand as overhead by adding an internal action to the model.

The source code of the QVT-o implementation and the code for the performance curves is available online in our git-repo[7].

7.7. Evaluation

In the following section, we evaluate the performance curves using a set of SPEC benchmarks. To do so, we describe the experimental setup and the method in the first part. Later we report on the results.

7.7.1. Method

To research the usability of the performance curves, we compare the performance prediction to the measurements taken from real executions. To cover a broad set of scenarios, we use SPEC benchmarks. SPEC offers three benchmark suites for parallel applications: MPI 2007, OMP2012, ACCEL. OMP2012 uses an OpenMP implementation of 13 different applications which cover a comprehensive set of application types. ACCEL focuses on GPUs, and therefore uses OpenCL implementations. MPI 2007 uses MPI as a means to parallelise and focus HPC systems. Thus, ACCEL and MPI2007 do not fit our domain, and we decide to use OMP2012.

To compare the measurements with the predictions, we first group the application within the benchmark suite according to the demand type we assume they have. Thereby, we use the documentation provided by SPEC. To give an example, the documentation of the benchmark suite `bt311` reads as following: *"BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations.*

[7] `https://github.com/PalladioSimulator/Palladio-Addons-ParallelPerformanceCatalogue`

The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension.”[8]. Because the characteristics are similar to the `MatrixMultiplication` demand, we assign `bt311` to the group of `MatrixMultiplication`. Table 7.4 shows the mapping of the benchmark applications to the expected demands.

Demand Type	Benchmark
`PrimeNumbers`	`botsalgn`
`MandelSet`	`smithwa`
`MatrixMultiplication`	`nab`, `bt311`, `fma3d`, `swim`, `bwaves`, `kdtree`,
`CountNumbers`	`md`, `botsspar`, `applu311`
`FibonacciNumbers`	`imagick`, `ildbc`
`SortArray`	

Table 7.4.: Mapping of benchmark applications to expected demand types

After the mapping, we execute all benchmarks on our hardware. Thereby, we increase the number of worker threads step by step, from one up to twice the number of physical cores[9]. Figure 7.11 shows the speedup curves for all benchmark applications within the benchmark suite. We can see that the maximum speedup for each application varies from 7 to 44. Further, we can see different behaviour characteristics for all applications.

[8] https://www.spec.org/auto/omp2012/Docs/357.bt331.html
[9] Unfortunately, we were not able to run the benchmark *ildbc* due to technical issues.

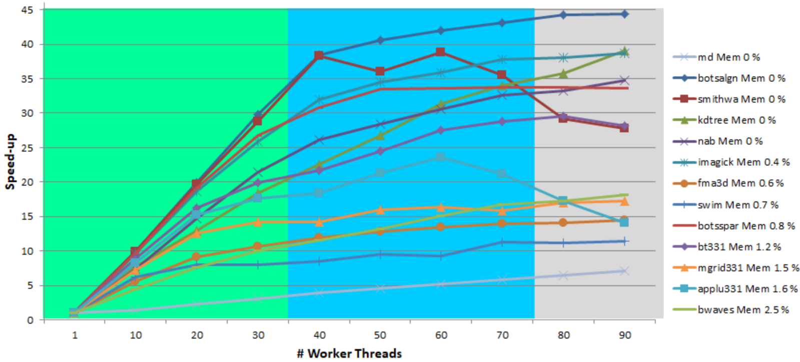


Figure 7.11.: Speedup Curves for the Applications from the OMP2012 Benchmark Suite

Next, we model the scenario using the parallel architectural template catalogue and the performance curves in Palladio. Our models consist solely of a single parallel loop and one internal action. We use the measurements from the sequential run to calibrate the CPU resource demand for the internal action, and specify the parameters of the parallel loop accordingly (e.g., number of worker threads, and demand type). In a final step, we compare the measurements from the execution with the simulation results.

Due to the extensive runtime of the benchmarks, we are only able to test one hardware setting. Therefore, we choose the more comprehensive system in Potsdam (40 physical cores—see Table 7.2), since it is a mid-range system and covers the characteristics of the smaller system and the machine in Stuttgart.

In the following, we present the results. To foster understandability and presentation style, we show only the accuracy of the predictions and not the actual runtimes. All measurements, simulation results, raw data, and performance curves are available online[10].

[10]https://doi.org/10.5281/zenodo.4081091

7.7.2. Results

In the following, we discuss the results of the experiment and the simulation. We will not present any raw data, but rather focus on the processes data. Table 7.5 shows the individual benchmarks from left to right. From top to bottom, the different number of worker threads are displayed. Further, we distinguish between the pure Palladio approach (top) and the Palladio approach using ATs and performance curves.

Each cell contains information about the inaccuracy of the approach. Thereby we compared the simulated runtime with the measurements and we calculate the accuracy difference as following:

$$P_{Error} = \frac{(t_{predictionTime} - t_{runtime})}{t_{runtime}} * 100 \tag{7.1}$$

The closer the number is to zero, the more precise is the prediction. For example, if we measure a runtime of $100ms$ and have a prediction of $80ms$ the prediction error is -20%. The minus indicates that the prediction is underestimated.

In our goal G_3, we aim for performance predictions that do not differ more than 40% of the actual measurements. Thus, we colour cells with inaccuracy below 40% green. As we see, the pure Palladio approach is accurate for a low number of worker threads. However, it becomes more inaccurate for higher numbers.

In contrast, the performance curves approach is able to satisfy our 40% limit in half the cases. Further, Table 7.6 shows the increase in accuracy compared to the pure Palladio approach. We calculated these values by the following equation:

$$\Delta P_{Error} = |P_{Error}(Palladio)| - |P_{Error}(PerfCurve)| \tag{7.2}$$

We can use simple subtraction to calculate the delta in the prediction error since the divisor is the same for $P_{Error}(Palladio)$ and $P_{Error}(PerfCurve)$. However, for the same reason, we can only compare the results within a column with each other and cannot compare values from different columns.

As we see, we can increase the accuracy significantly for a large number of worker threads (more than ten) and even for a low number we are in eight out of twelve cases more precise. Additionally to the tables, we provide a visualisation of the values for the best and the worst scenario in Appendix A.4.4.

Please note that the values in Table 7.6 are only intended to show in which cases the performance curves perform better and in which cases they perform worse than the pure Palladio approach. Due to the nature of relative values, and the fact that each column has a different divisor, a comparison of the values is not valid.

Overall, the measurements show that the use of performance curves dramatically contributes to the accuracy aspect of performance predictions. However, they also show that we have not yet captured all PPiFs. Especially demands which show a low speedup and thus are bad to parallelise are ultimately not captured in the performance curves. Identifying additional PPiFs, measuring their influence, and deducting more precise performance curves is still an open challenge and remains for future work.

At this point, we can present a total of twelve performance curves which already greatly improve the performance prediction capabilities of tools like Palladio. Further, we provide integration into Palladio. Thus, we enable the SA to efficiently use the performance curves and benefit from more accurate prediction results.

7.8. Assumptions & Threats to Validity

To conclude our results from the evaluation and to put the results in perspective, we discuss assumptions made and threats to validity in the following. Therefore, we list each assumption or threat and discuss it in detail.

Monitoring Overhead: During the execution of all experiments, we monitored only one PPiF at a time (e.g., response time, L1, L2, L3, etc.). Thereby we use different tools to monitor the runtime (e.g., perf or PAPI). The usage of these tools puts overhead on the system and might influence performance factors, or even have an impact on the

	Threads	Benchmark imagick	botsalgn	smithwa	nab	bt311	fma3d	swim	bwaves	kdtree	md	botsspar	applu311
Palladio	**10**	-3,96	-0,40	-0,81	-26,12	-11,32	-44,27	-38,23	-56,14	-22,01	-86,47	-17,53	-17,53
	40	-20,16	-3,86	-4,30	-34,67	-45,65	-70,22	-78,63	-71,17	-43,60	-90,22	-22,84	-53,88
	80	-52,50	-44,71	-63,57	-58,46	-63,12	-82,32	-86,02	-78,37	-55,33	-91,97	-57,82	-78,40
	90	-51,59	-44,48	-65,33	-56,48	-64,87	-81,86	-85,66	-77,29	-51,22	-91,18	-57,93	-82,44
Perf **Curve**	**10**	6,24	10,91	8,29	-10,33	7,62	-32,37	-25,03	-46,77	-5,36	-84,56	-5,85	-5,85
	40	-11,69	7,06	4,48	-20,71	-34,04	-63,86	-74,06	-65,01	-31,56	-88,83	-11,92	-47,35
	80	1,29	5,31	-56,32	-6,22	-16,75	-60,09	-68,43	-51,17	0,84	-65,07	83,39	-6,07
	90	4,41	2,97	-57,31	1,06	-18,42	-57,87	-66,70	-47,27	13,28	-60,19	89,98	-20,72
	Curve	Fibo	Prime	MandelSet	Matrix						CountNumbers		

Table 7.5.: Shows the inaccuracy of the pure Palladio approach in comparison to the Palladio + Performance Curve approach for the SPEC OMP2012 Benchmark set.

Threads	Benchmark imagick	botsalgn	smithwa	nab	bt311	fma3d	swim	bwaves	kdtree	md	botsspar	applu311
10	-2,29	-10,51	-7,48	15,78	3,70	11,90	13,19	9,37	16,66	1,92	11,67	11,67
40	8,48	-3,20	-0,18	13,95	11,61	6,36	4,57	6,16	12,05	1,38	10,92	6,53
80	51,21	39,41	7,25	52,24	46,37	22,23	17,58	27,20	54,49	26,90	-25,57	72,33
90	47,17	41,52	8,02	55,43	46,45	23,99	18,96	30,03	37,94	31,00	-32,05	61,72

Table 7.6.: Shows the accuracy gain of the performance curve approach in comparison to the pure Palladio approach

PPiFs under review. Thus, we were able to observe higher runtime and worse speedup when monitoring memory behaviour.

Memory Bandwidth Even though we acknowledge the importance of memory bandwidth, we did not measure the throughput and utilisation of the memory bus, due to the challenging character.

Interdependencies: When analysing the results, we only looked at one PPiF at the time and neglected interdependencies, although we are aware that this might be a naive assumption.

Synthetic Demands: We choose in favour of synthetic demands, because synthetic demands are easier to handle, and thus, they are suited to researching PPiFs. However, their behaviour differs from real demands, especially for medium numbers of worker threads. Therefore, they are not perfectly suited to extract performance curves from. Even though we achieved promising results when pulling performance curves from the synthetic demands, future research has to look into the use of real demands.

Hardware: We used four different kinds of hardware environments. Even if we try to have a homogeneous test environment, we make observations that we currently cannot explain, and are bound to the hardware. Further experiments on different hardware environments might help to gain additional insights.

Use Cases: We evaluated our performance curves against the SPEC performance benchmarks. However, to thoroughly verify the performance curves, we need to assess them against real, or rather, business use cases.

High Abstraction: The use of performance curves adds additional load to the performance models. Thereby, the load is very abstract and does not map to specific PPiFs. Thus, even if we achieve better predictions, the explainability of the models suffers.

Overhead Modelling: During the integration of the performance curves into Palladio, we used the overhead function modelling approach from the parallel patterns. This approach adds additional demand (i.e., CPU demand) to the model to emulate the overhead. So the CPU shows a higher utilisation. In reality, this might not be accurate,

because the overhead could also come from waiting conditions. Thus the simulations might show higher CPU utilisation than the actual system.

Over-interpretation of Results: In Table 7.6, we indicated the accuracy gain of the performance curves in contrast to the pure Palladio approach. Thereby we subtracted the relative values in Table 7.5, which is theoretically possible because the divisor of both values is the same. We decided to display the values in Table 7.6 in this way, to give an impression of the number of cases in which the performance curves perform better. However, a comparison of the values from different columns would lead to wrong or over-interpretation of the results, because each column has a different divisor (see. Section 7.7.2).

7.9. Summary of CB_2

In this chapter, we researched performance curves for parallel applications in multicore environments. Thereby, we worked on the fulfilment of requirements $R_{modelling}$, $R_{performance}$, and $R_{accuracy}$.

In the course of the chapter, we first performed a structured literature review in combination with expert interviews to identify the most relevant PPiFs. Next, we conducted extensive experiments to (a) evaluate the impact of PPiFs on performance and (b) collect measurements to extract performance curves. During the experiments, we researched different hardware environments and parallelisation paradigms as well.

As a result, we present 14 lessons learned from the experiments. Additionally, we deliver a set of twelve performance curves to the SA. The performance curves represent the most relevant software behaviours. Combining the performance curves with performance prediction approaches such as the PCM, we show that the accuracy of parallel application predictions increases greatly. Thus, we provide an instrument to the SA that helps to improve accuracy of model-based performance predictions on an architectural level for parallel applications in multicore environments.

To evaluate the performance curves, we use a standardised benchmark suite—SPEC OMP2012—and compare the predictions from Palladio (containing the

performance curves) with the measurements we took from executing the benchmark on a medium-sized multicore environment. We show that the performance curves increase the accuracy for all cases in which we use a high number of worker threads (equal to the number of virtual cores) and, in 19 out of 24 cases, of a low number of worker threads—when compared to the default Palladio approach.

In a nutshell, we are able to answer our research question as follows:

$RQ_{2.1}$: **How do highly parallel applications behave in massive parallel environments (multicore systems) regarding response time (speedup), memory access rates (L1, L2, L3, RAM usage), and memory bandwidth utilisation?**

Answer: *In over 800 experiments we took 70,000 measurements. Thereby, we monitored the response time and memory accesses of the systems. Using these measurements we extracted the twelve performance curves given in Table 7.3 to describe the behaviour.*

$RQ_{2.2}$: **What factors influence performance the most in highly parallel applications?**

Answer: *In Table 7.1 we listed the top eight performance-influencing factors we identified via structured literature review, expert interviews, and our experiments.*

$RQ_{2.3}$: **Does the choice of parallelisation strategy have a significant impact on behaviour?**

Answer: *The experiments show slight differences in the performance of the individual parallelisation paradigms. However, these differences are not significant for all thread-based paradigms. The only paradigm that diverges is the AKKA Actors implementation. Here we assume issues in the framework coding.*

$RQ_{2.4}$: **Do highly parallel applications show similar behaviour, which can be described by one or multiple performance curves?**

Answer: *In Table 7.3 we present performance curves for all the research resource demands. We used linear regression to extract the curves from the measurements. Thus, the curves describe the average behaviour for each demand type on all the tested machines.*

$RQ_{3.2}$: **What are the missing characteristics of software behaviour that must be included in performance prediction models (performance-influencing factors) to enable simulation-based performance prediction approaches to accurately predict the performance of parallel applications?**

Answer: *Table 7.1 shows the top eight most performance-influencing factors, gained from structured literature reviews, expert interviews, and experimenting.*

Finally, we can verify or falsify our hypothesis as follows:

$H_{2.1}$: **The speedup and performance behaviour of highly parallel applications depends heavily on the chosen parallelisation strategy or paradigm.**

Reject: *The choice of parallelisation strategy does not have a high impact on behaviour.*

$H_{2.2}$: **The hardware architecture (e.g., number of CPU cores, memory bandwidth, memory hierarchies) of the execution environment has a strong impact on the performance of the parallel applications.**

Accept: *We measured differences in the normalised speedup for all the machines. Thus, we can verify that the hardware architecture has an impact on the performance. The biggest noticeable difference is*

between virtualised hardware and dedicated systems. Virtualised hardware shows worse performance.

$H_{2.3}$: **The speedup of a parallel application is not only influenced by the number of cores available in a system but also by additional hardware specific performance-influencing factors.**

Accept: *In Table 7.1 we listed the top eight performance-influencing factors we identified.*

8. CB_3: Meta-Model Extension for the PCM to Include Memory Architectures

Single-metric hardware performance models, which only consider CPU speed as a relevant characteristic, have proven insufficient. Therefore, we research the effect of additional metrics like memory architecture, hierarchies, and bandwidth in this chapter and focus on the requirements $R_{metrics}$, $R_{performance}$, and $R_{solvers}$.

In the previous chapter, we researched the influence of worker threads, cores utilisation, resource demand type, and parallelisation paradigms. Thereby, we observed the cache behaviour and cache access. In this section we continue to research the next PPiFs: memory design and memory bandwidth (see Tab 7.1).

To do so, we extend the PCM, adopt the solvers, and update the editors of the Palladio bench. Thereby, we follow the research process illustrated in Figure 8.1.

In the course of this chapter, we first define the problem space and the research goal, introduce the idea behind the approach, and set the evaluation criteria. Next, we research the problem space of memory hierarchies to identify relevant elements to include in the meta-model. Afterwards, we discuss meta-model extension strategies and perform the extension. To support the new meta-model features, we extend the editors (tree-editor and Sirius-editor) and the simulators (SimuLizar). Finally, we evaluate the approach in an experiment-based manner and compare the new performance predictions to the earlier ones without consideration of memory bandwidth.

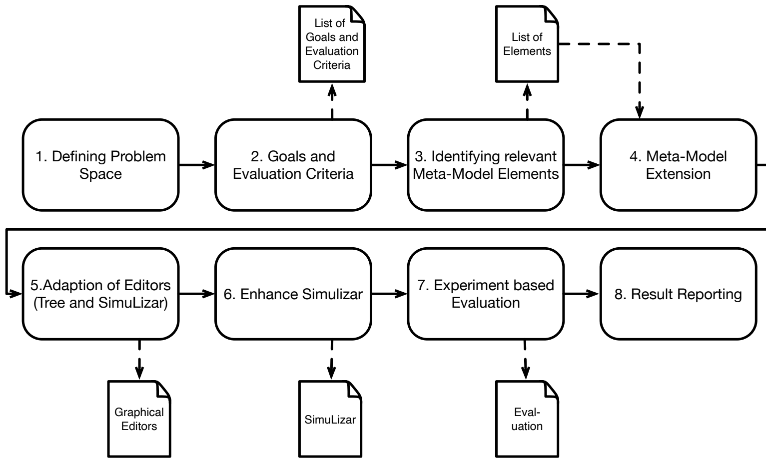


Figure 8.1.: Overview of the Research Method for Contribution CB_3

As a contribution we (1) give detailed insights into the behaviour of parallel applications, (2) provide a meta-model extension for the PCM, which included memory hierarchies, (3) provide a SimuLizar extension to simulate memory hierarchies, and (4) lay out four modelling approaches with different strengths and weaknesses.

As a result, we can show that the four memory model approaches increase the performance prediction accuracy of Palladio. Each model works exceptionally well under certain circumstances. Overall, we present the cache-line model, which has the best overall performance prediction power and increases the accuracy of up to 57%. Thus, in the best case, the prediction error is below 15%.

Significant results of this chapter were acquired while collaborating on the supervision of student theses by [Gru19] and [Tru20]. Further, insights from the first two steps of the research method were reviewed and published in [GF19].

We make all data, meta-models, code, and plugin extensions online available:

Section 8.3 Meta-Models and Palladio Plugin:
`https://github.com/PalladioSimulator/Palladio-Addon-MemoryHierarchy`

Section 8.4 Evaluation and Results:
`https://doi.org/10.5281/zenodo.4094588`

8.1. Problem Space

In this chapter, we research the impact of memory architectures of a multicore CPU on the overall performance. Thereby, we follow the hypothesis that for modern complex multicore CPUs, not only the clock rate but also the memory bus is a bottleneck. Further, we assume that the sizes and utilisation of the caches have a significant impact on the overall performance of highly parallel applications [BDH08; BKR09; FBKK19; FH16].

Prototype—Using Network-Links as Memory Bandwidth Model: In [GF19], we research the impact of a simple memory model by using network links to emulate the data transfer. By observing and measuring the memory utilisation of a real application, we were able to calibrate the model and to increase the performance predictions up 26% for a 16-core machine.

The insights from the prototype encourage us to further investigate memory hierarchies and to properly include them into the PCM.

Research Goal and Idea: The research in this chapter focuses on the fulfilment of two goals:

G_1**:** SAs shall be able to model memory hierarchies in the hardware model and specify the memory access behaviour in the software model.

G_2: Given these additional model elements, the solver shall give more accurate performance predictions for parallel applications.

Since we know that the memory bus has a great impact on the performance [GF19], we aim to include a concept very similar to network links for the memory bus. However, we need to face and overcome the following challenges:

C_1: We need to identify further PPiFs for memory architecture via a literature search.

C_2: Given the PPiFs, we need to determine required meta-model elements and include them into the meta-model.

C_3: The PCM solvers need to be adapted to be capable of interpreting the new meta-model elements.

Research Method: To achieve our goals, we continue to follow the experiment-based performance model derivation method [Hap08] and iteratively extend the meta-model. To evaluate the results, we compare the current PCM and solvers with the extended ones and compare the simulation results to the results from the experiments for our running use cases. Thereby, we only focus on a single evaluation criterion:

E_1 : The accuracy of the new performance predictions needs to be better than the current ones—better meaning closer to the real measurements from the experiments.

8.2. Meta-Model Extension

In the following, we describe the meta-model extension in detail. To achieve the final model, we follow the method of experiment-based performance prediction [Hap08]. That way, we go through the process of modelling four times, adapting solvers and editors, and evaluating [Tru20] the results. Nevertheless, we describe only the final result in the following.

First, we will research the required model elements. Next, we discuss different strategies to extend a meta-model, along with their advantages and disadvantages. Finally, we describe the changes to the PCM in detail.

8.2.1. Meta-Model Elements

To identify the required model elements, we use the identified PPiFs (see Table 7.1) as a starting point and choose to include caches, main memory, and the memory bus. At first, it seems reasonable to include all PPiFs, along with all attributes, and thus have a model which is as close as possible to the real-world objects. However, having such a meta-model would increase the complexity by far and the SA would not be able to handle the architectural design.

Thus, we follow the general definition of modelling by [Sta73] and the goal-driven modelling approach by Koziolek for qualitative modelling [RBH+16]. Therefore, we define the following three properties for our modelling approach:

Pragmatism: Currently, Palladio simulations for multicore CPUs result in a linear speedup correlating to the number of cores. However, real executions show non-linear speedup. Therefore, our goal is to model the memory behaviour on an abstract level to capture the performance-relevant factors and represent the non-linear speedup behaviour, or at least parts of it.

Representation: Models are always the representation of something, i.e. a mapping or representation of natural or artificial originals—it can be a model itself. In our case, we want to represent the performance-relevant attributes of memory architectures. Thus, we focus on the timing-related aspects and not on the resource demands, such as memory utilisation.

Reduction: Describes properties that can be simplified or ignored in the model. In our case, all memory hierarchy attributes that do not contribute to the pragmatism should be neglected. Following C_2, the challenge is to identify negligible attributes. Here we rely on literature and experiments to determine these negligible attributes.

In the following, we describe each of the PPiFs we consider, and give the first set of relevant attributes for the meta-model:

Cache: Modern multicore CPUs have multiple caches on different levels (see Section 2.2). Modern CPUs have L1, L2, and L3. However, in our performance prediction models, we consider arbitrary cache levels. Whenever an application requires data, the cache is hierarchically queried until the required data is found. In the worst case, the main memory needs to be read. For each cache we define the following:

Size: The cache sizes are an important factor for the cache effectiveness. However, considering the size for performance prediction would lead to a full cache simulator, which can easily become very complex, not only to implement, but also for the SA to specify. Therefore, we decided not to consider the cache size in our models, but to focus on the cache hit or miss rate. Thus, we abstract the cache behaviour.

Hit-rates: The cache hit-rate gives the probability that a cache request will be fulfilled (e.g., 40%). In case a cache hits, we assume an immediate delivery of the results with no delay. In case of a cache miss, the next cache has to be queried, and the cache updates its cache page, which puts additional demand on the bus.

Page-size: The size of the cache page is relevant to specify because in case of a cache miss, the whole cache page will be updated and needs to be fetched from the main memory. Thus, each cache miss puts additional demand on the memory bus.

Type: Caches can be shared or private. Common architectures have private L1 and L2 caches, while the L3 cache is shared [Sch08]. Only a single core can access private caches. Shared caches can be accessed by multiple or all caches.

Main Memory (DRAM): The main memory is the last point of access if all previous caches fail to provide the required data. For our model, we assume that the size of the main memory is infinite and that all data are available.

Memory Bus: The memory bus interconnects the CPU cores, L1, L2, L3, and the main memory. In our model, we assume that the bus always connects two parts (e.g., L2 and L3). To determine the performance characteristics of the memory bus, we use the following attributes.

Latency: Memory latency is the time between initiating a request for data and the beginning of the actual data transfer. In our models, we neglect the latency, because we assume that latencies are very low and do not have a major impact on the overall performance. However, we include it in the meta-model for use in the future.

Bandwidth /Throughput: Describes the maximum throughput of the bus being fully utilised—burst rate (e.g., 12 GB/s).

Dynamic: Due to the architecture and composition of cores, caches, memory, and bandwidth, the maximum throughput of the memory bus can vary according to the number of cores used. In general, the overall throughput increases with usage of more cores, due to additional resources (e.g., buses) becoming available. This is especially true if a new NUMA node is utilised (see Section 2.2). Since we do not consider unique architectures like ccNUMA, but want to provide an abstract model, which the SA can use for all kind of architectures, we need to provide an abstract attribute to specify this behaviour.

Composition: The composition of the elements mentioned earlier is a critical factor, and we need to consider the composition of all elements in the meta-model. Thus we assume that cores, caches, and main memory can be connected via a memory bus to arbitrary architectures.

To clarify the choice of PPiFs and attributes and to further follow the argumentation line, we have to explain a set of assumptions we made.

L1: L1 is usually divided into instruction and data cache. However, we assume that the impact is rather low, so we ignore the operation and handle L1 as a normal cache.

NUMA: Multicore CPUs differ in NUMA nodes (see Section 2.2). The access time from a CPU to a memory element within the NUMA node is faster than accessing data located in another NUMA node. Since we

consider the modelling of NUMA nodes too complex, we ignore these effects for now. Later we will have to re-evaluate this decision.

Reading time: We do not consider the cache or memory access times or the latencies of the caches or main memory.

Swap Operations: As stated above, we only consider timing effects and no memory utilisation. Thus, we assume the main memory to be infinite. However, this is not true for the real system and can have a huge impact, especially for memory-consuming applications on systems with comparatively low memory. In such environments, it might happen that the memory size is not enough to store all the required data; if that happens, the memory controller stores data on the hard drive and swaps data between memory and hard drive if needed. Since access to the hard drive is a lot slower, this can have a performance impact. Nevertheless, we ignore this scenario for now, because we consider it to be an exception, since modern architectures come with a large amount of main memory.

Complex Cache Behaviour: The behaviour of caches follows complex rules, including invalidation cache pages, synchronous reads, and keeping cache coherence. For example, it is possible to have the same data in two different L2 caches. If one value is changed, the cache page of the other cache needs to be invalidated and the cache page needs to be updated. The memory controller ensures cache coherence. This is a complex process, which (when addressed in the models) would result in complex models. To keep the models simple, we do not consider this behaviour for now.

Reads & Writes: The typical operations on the memory are reads and writes. Both operations have slightly different latencies [Gru19]. However, for now, we consider them as equal and do not distinguish between the operations.

8.2.2. Meta-Model Extension Strategies

Before we start to model the above elements into the meta-model, we first need to discuss extension strategies. In general, there are two possible ways to extend the PCM.

The first is a full PCM extension. In a full meta-model extension, we model the changes and new elements directly into the PCM. After altering the PCM, we need to release a new version. Advantages of this approach are that all model elements are in one place, and it is straightforward and easy to follow. However, on the downside, a new release of PCM has a long-range impact. For example, we cannot guarantee that all solvers and tools can handle the new version.

Therefore, we favour a second approach, in which we use Profiles and Stereotypes [FV04] to extend the PCM. That way, we can model our memory model and all elements in a separate model. Using profiles and stereotypes, we can link our new model elements into the PCM. The advantages of this approach are that we do not need to release a new PCM version, but can provide the memory models and profiles as a plugin. Solvers and tools which cannot handle the new elements ignore those. Thus, we can guarantee downward compatibility. On the downside, this approach becomes unusable if we need to alter many already existing elements. However, in our scenario, this is not the case.

8.2.3. Hardware Model Extension

In the following, we detail the extension of the meta-model. For this, we first look at the extension of the meta-model to enable the SA to model the hardware characteristics in the hardware model. To do so, we pick an entry point and lay out the meta-model. In the next section, we explain the changes to the workflow and adaptations of the software model. In the software model, the SA needs to specify the memory behaviour, e.g., memory accesses.

8.2.3.1. Entry Point

Given the current version of the PCM (version 4.2.0), we identify multiple elements which we can use as an entry point for the extension:

ProcessingResourceSpecification: The `ProcessingResourceSpecification` contains the information on the processing resources. This entry point is suitable because we can reuse the predefined resources CPU,

HDD, and Delay and add our processing type for the memory hierarchy. However, to fully support the characteristics of memory hierarchies, we need to add elements for the hierarchical structure. Further, a `ProcessingResourceSpecification` requires a processing rate and a scheduling policy, which do not apply to memory elements. To avoid ambiguity in our models, we decided against the `ProcessingResourceSpecification`.

ResourceContainer: The resource container is the more general model element. It can contain a `ProcessingResourceSpecification` and other hardware-related characteristics. From the modelling aspect it has no disadvantages. Thus, we choose it as an extension point.

As described in the previous section, we choose a profile-based extension strategy. Given the `ResourceContainer` as starting point, we can now start to model our meta-model extension. Figure 8.2 shows the applied profile to the `ResourceContainer`. It maps our meta-model extension (MemoryHierarchyMetamodel) into the already existing PCM element (ResourceContainer). In the next section, we explain the memory meta-model mapped into the resource container.

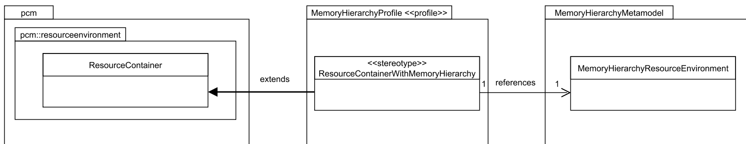


Figure 8.2.: Overview of the Profile Extension of the ResourceContainer

8.2.3.2. Modelling the Memory Hierarchy

Design Rationale During the design of the memory meta-model, we follow the Palladio design principles and approaches. Therefore, we align our modelling to existing elements or reuse them if possible. This brings two benefits: First, the SA is familiar with the modelling concept; second, the simulation logic of existing elements can be reused or adapted only slightly.

When analysing the PCM (Version 4.2), we identified two elements which we can reuse:

ResourceContainer: A `ResourceContainer` represents a server. It can be specified with multiple `ProcessingResourcesSpecifications`, which are further specified as a `ProcessingResourceTypes` (e.g., CPU, HDD, or Delay). These three `ProcessingResourceTypes` are currently predefined in Palladio. However, it is also possible to add additional resource types (e.g., memory). Moreover, for `ProcessingResources-Specifications` it is possible to specify the processing rate (e.g., CPU cycles), the scheduling policy (e.g., processor sharing), and the number of replicas, which can be used to specify the number of CPU cores on a server. As described above, the `ResourceContainer` is our entry point, and therefore we will reuse it as it is.

LinkingResource: A `LinkingResource` represents network links. Network links connect ResourceContainers. The `LinkingResource` can be specified with `CommunicationLinkingSpecifications`, which can have different `CommunicationLinkResourceTypes`, similar to the `Processing-ResourceType`. Palladio also offers the predefined LAN `Communication-LinkResourceType`. Furthermore, it is possible to specify throughput and latency in the `CommunicationLinkingSpecifications`. This concept is very close to the memory bandwidth. Thus we reuse it to model the memory bus.

In the following, we introduce the meta-model extension. Thereby, we focus only on the extension part—the memory architecture.

Memory Meta-Model Figure 8.3 shows the final meta-model extension. At the top of the figure, the `MemoryHierarchyContainer` represents the top-level element and the entry point. Each container can have multiple `Memory-HierarchyResourceEnviroments`. Each environment consists of multiple memory elements (i.e., caches or main memory) and a set of connections (i.e., the memory bus). Further, each environment has an entry point, which defines the entry point of the memory architecture. Both the starting point and the memory element are of the type `LinkableMemoryHierarchyResources`.

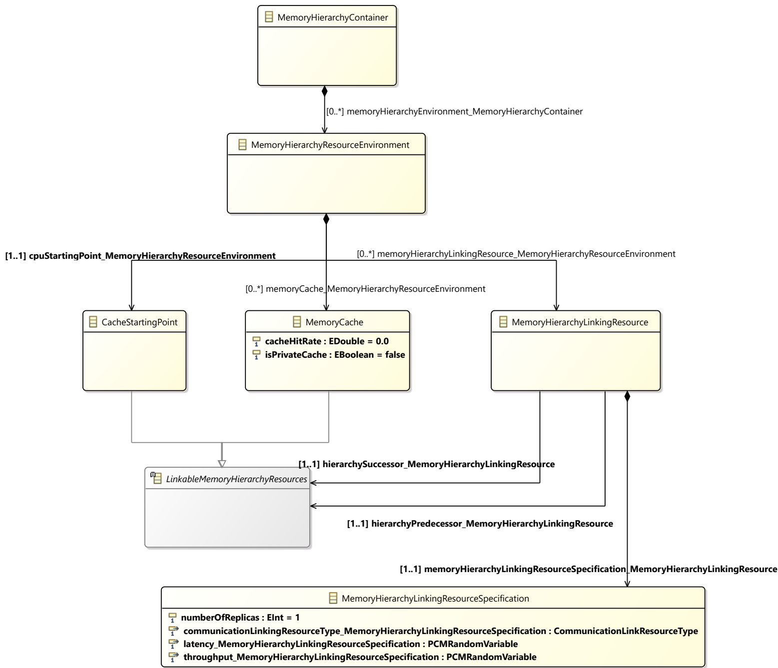


Figure 8.3.: Meta-Model Extension Containing the New Elements for the Memory Hierarchy

We decided on this way of modelling because (a) it aligns with the network link layout, and (b) we are more flexible for extensions and further adaptations.

The memory element has two attributes. The `cacheHitRate` describes the possibility of a request to result in a cache hit. The `isPrivateCache` defines whether the cache is private or shared by other elements in the architecture. A memory element is connected to another element via a `MemoryHierarchy-Linking Resource`. A linking resource always connects two linkable memory resources—one as successor and one as the predecessor. Further, the linking resource has a specification.

The MemoryHierarchyLinkingResourceSpecification has in total four attributes, which are all adapted from the network linking resource. The number of replicas defines the total number of busses available. The latency describes the time between the initial request for data and the data transfer; the throughput describes the maximum data transfer capacity of the link.

With this extension, the SA is now able to specify the memory characteristics in the hardware model. Next, the SA needs to specify the memory behaviour in the software model.

8.2.4. Modelling Memory Behaviours

To utilise the memory architecture specified in the hardware model, the SA has to set the memory behaviour in the software model as well. In the following, we discuss the extension of the software model and the adaptation of the workflow.

8.2.4.1. Resource Demanding Calls

To specify resource demands (e.g., CPU or HDD or memory demands) the SA requires a model element that can name specific resources. In the PCM these elements are named Calls and are specified in the SEFF (see Section 2.4.2.1). For the memory resource demand, we evaluate existing calls to check their reusability. In total, we evaluate six call actions:

ExternalCall: The external call is used to specify the communication between components. In contains information about the parameters passed to an API and the data size. Since the external call always references an OperationRequiredRole, which is the interface specification of the calling component, this call is not suitable for our purposes. Even though we could allow the SA to specify the OperationRequired-Role in the sense that he can set the memory level directly, the SA should not set the memory hierarchy manually.

Acquire/ReleaseAction: Acquire and release actions are used in the PCM to allocate passive resources. Willnecker et al. [WBKK15] used passive resources to simulate the memory demand for garbage collection

in Java applications. However, since we are not interested in the memory consumption of the system, but more in the delay generated by memory architectures, we do not consider these calls further.

InfrastructureCall: Infrastructure calls introduced by [Hau09] behave similarly to external calls, but these calls also represent architectural levels. Thus, infrastructure calls are used when calling a lower-level component that runs on the same hardware. It is not suitable for the same reasons that we reject external calls.

InternalAction: The internal action represents an action within a component, e.g., a code instruction. Each action can have resource demands like CPU or HDD. For our memory model, a specification of memory demand here makes sense as well.

InternalCallAction: The InternalCallAction is used to model nested `ResourceDemandingInternalBehaviour`, e.g., a Java method that has several private sub-methods. We did not consider this call further, since it does not give additional befits for us and is not supported by the current simulator versions and editors[1].

ResourceCall: The resource call is another call that enables us to call resource demands. In contrast to internal action, the resource call allows a fine-grained specification of the resource demands. For example, it is possible to define different demands for reading and writing operations for HDD. Due to this characteristic, the resource call is best suited for specifying memory behaviour, because it might become necessary to separate read and write requests.

Given the evaluation of the discussion, we have to choose the abstraction level on which we want the SA to model the memory behaviour:

low: If we want the SA to specify the memory demand on a low level, along with the specific cache to access, the infrastructure call will be the best choice.

medium: If we want the SA to specify the memory demand more abstractly but still allow us to separate between reading and writing operations, the resource call will be the best choice.

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

high: If we want the SA to specify the memory demand abstractly and only enable her to define the demand in a parametric manner, the internal action is the best choice.

Because we are convinced that the separation of reads and writes is essential when researching the performance impact of memory architectures, we chose in favour of the resource call.

8.2.4.2. Integration of Memory Calls into the SEFF

In the current version of Palladio, it is not possible to modify an existing `resourceCall` action to handle a customised behaviour inside the simulation, so we have to implement a workaround. Thus, we use the chid-extenders (or sub-classing)[2], which we can use non-invasively (e.g., not changing the PCM) to create a clone of the `resourceCall`, which we use within the simulation to implement our custom behaviour. At the same time, we propose a code change[3], which enables the customisation of resourceCalls. Therefore, this should be just a temporary solution.

8.3. Adaptation of PCM Solvers

Enabling the SA to model the memory hierarchies in the hardware model and the memory behaviour in the software model is only part of the solution. In the next step, we need to adapt the PCM solvers so that they can interpret and analyse the new model elements. Palladio contains a number of different solvers (see Section 2.4.2.1). In the following, we briefly describe the adaptation of SimuLizar, the current default simulation-based solver. We give only a high-level description of the process and implemented behaviour. We do not give detailed information on the implementation. For this we refer to [Tru20] and the code[4].

[2] https://ed-merks.blogspot.com/2008/01/creating-children-you-didnt-know.html

[3] https://jira.palladio-simulator.com/projects/SIMUCOM/issues/SIMUCOM-97?filter=allopenissues

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

To realise the recognition of the memory hierarchy, we use the observer extension point. All observers, like `ResourceEnvironment`, `ResourceContainer`, `NetworkLinks`, and `ProcessingResources`, are called, and representative Java classes are created. These Java classes are all stored into the model registry class that can be used to look up and access these elements during the simulation.

The `PCMStartInterpretationJob`—which is the simulation entry point—inside SimuLizar consists of two phases: (1) set up and (2) simulation. During the set-up, the initialise () method of all classes that use the model observer extension point is called. In this phase, the `MemoryHierarchyObserver` class looks for `ResourceContainer` elements that have the `ResourceContainerWithMemoryHierarchy` stereotype applied. Next, it searches, creates, and stores objects representing the modelled memory hierarchy structure into a `MemoryHierarchyRegister` class. The `MemoryHierarchyRegister` stores all necessary information about the memory hierarchy structure. Therefore, we use the register to look up all the required memory hierarchy information during the simulation. In the simulation, the memory demand is specified with the `InternalActionWithMemory` model element, which is a subclass of the `InternalAction` and has no difference from the `InternalAction`. The only difference is defined in the memory hierarchy ecore model—not the PCM ecore model. That way, and with the help of the rdseff-switch extension point, which can delegate the interpretation of a call that is not inside the SeffPackage to other plugins that support this extension point (e.g., the `InternalActionWithMemory`), the call is delegated into the `MemoryHierachyCallAwareSwitch` of the memory hierarchy. Next, we can process the call here as we desire. In short, the `MemoryHierarchyObserver` is used to search for `ResourceContainer`, which contains memory hierarchy elements. Additionally, this class is responsible for creating necessary objects for simulation and for storing them in the `MemoryHierarchyRegistry`. During the simulation, the memory demand is reduced based on hit/miss-rate, and the updated demand is then simulated through the next `MemoryHierarchyLinkingResource`. Unfortunately, we cannot reuse `NetworkLink` implementations, because SimuLizar has no support for them. Thus, we use the NetworkLink code from SimuCom to implement The `MemoryHierarchyLinkingResource`.

In contrast to NetworkLinks, the `MemoryHierarchyLinkingResource` does not do round trips. To model the memory hierarchy, each core has its link

to the L1 and L2. For example, if we consider a 96-core server, a total of 192 linking objects are created during the simulation phase. To guarantee performant simulations, we added a modified version of the FCFC-scheduler, which can simultaneously handle multiple instances, instead of only one at a time.

8.4. Adaptation of Modelling Editors

While the default Eclipse tree editors are part of Eclipse EMF and provide an out-of-the-box approach to edit the memory model, we aim to include the modelling in the Palladio workflow. Because Palladio is using Sirius[5] to visualise the PCM graphically, we need to adopt the Sirius editors as well. In the following, we briefly describe the changes we made. Thereby we follow the Palladio style guides[6].

To extend the editors, we create two new plugin projects. One contains the .odesign file. The other has additional Java code to perform more complex actions.

In the .odesign file we have to specify two elements (see Figure A.22): The graphical elements and the tools. The graphical elements contain the nodes and edges. Here we define e.g., the memory cache element and the memory predecessor and successor link. The tools define the action the editor can perform on the model elements, e.g., double click, creating new elements, etc. (see Figure A.23).

Additionally, we use external Java actions to provide more complex editor features. For example, we use a dialog view to let the SA specify the throughput with the help of a stochastic expression (see Figure A.24).

To use the extended editor and diagram, the SA needs to enable the correct viewpoint (i.e., `SeffWithMemoryHierarchy`).

[5] `https://www.eclipse.org/sirius/`

[6] `https://sdqweb.ipd.kit.edu/wiki/PCM_Development/Sirius_Editors`

8.5. Evaluation of PCM Extension

To assess the usability of the memory model extension, we use an experiment-based approach based on the matrix multiplication example (see Section 5). Figure 8.4 gives an overview of the evaluation approach.

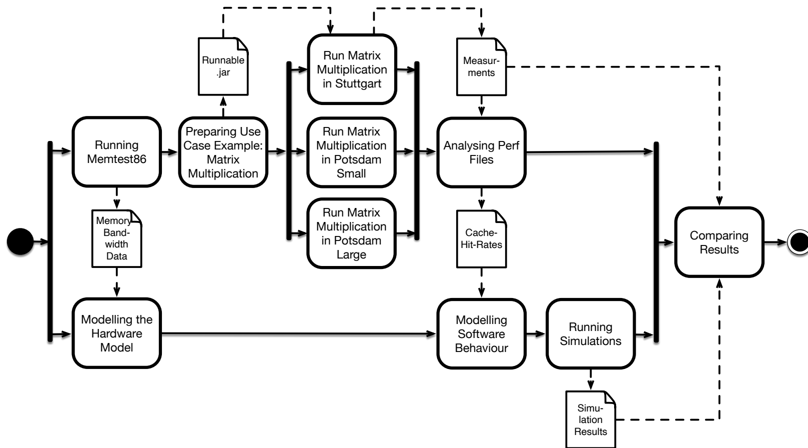


Figure 8.4.: Overview of the Evaluation Process for CB3

In a first step, we execute Memtest86[7] on our target hardware. This way, we get information about memory bandwidth, which we use to calibrate our performance model (i.e., the hardware specification). Next, we implement and execute the matrix multiplication example in our test environment. Thereby, we execute a different version with different matrix sizes. In this step, we also monitor the cache hit rates, which we use to calibrate our models further. In a common performance prediction process, the SA does not have this information at hand and needs to estimate the cache hits. But since we want to evaluate the performance models, we decide to use the information at hand to reduce the error-proneness. Finally we simulate the models and compare the measurements with the predictions from the simulations.

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

Since we are following the method of experiment-based performance prediction [Hap08], we iterate multiple times over the process of performance model creation. In total, we have four iterations, and in each one we create a performance model with specific properties. We present and discuss all models in the course of the evaluation.

We provide all results, raw data, and performance models in an online repository [8]. Further, we provide the extension as a Palladio plugin [9].

8.5.1. Experiment Setup

To set up the experiment, we first implement the matrix multiplication use case given the characteristics in Chapter 5 and the implementation we used in [FH16]. To parallelise the application, we use Pyjama [GS13], as we did in the previous chapter. We decided against using the synthetic demands from ProtoCom, because this time we want to provoke as many inter-thread communications as possible.

We executed the implementation of the three dedicated systems described in Table 7.2. We did not use the BWUniCluster. Due to the virtualised environment, it is not possible there to run perf or collect the performance properties we need for calibrating the model.

On each system, we perform multiple runs of the experiment. In each run, we change the number of worker threads, starting with one (sequential run) and increasing the number stepwise, up to twice the number of physical cores. Additionally, we consider two different matrix sizes. In the first scenario, we multiply matrices with a dimension of 3000x3000. In the second scenario, we consider a more massive matrix of 7000x7000. We use this scenario to guarantee that a matrix does not fit into caches. The system in Stuttgart has a particularly large cache space, so to force main memory accesses, we use larger matrix sizes.

For each configuration and scenario, we execute multiple runs (100 for the small and 50 for the large scenario) to eliminate variances and side effects. Due to the low standard deviation, we only consider the mean value in

[8] https://doi.org/10.5281/zenodo.4094588
[9] https://github.com/PalladioSimulator/Palladio-Addon-MemoryHierarchy

the following. Further, we recorded all performance counters during the execution using perf.

8.5.2. Model Calibration

For modelling and simulating the use case we use PCM nightly version (pre-release PCM 4.3.0), Eclipse 2019-09 Modelling Tools, and OpenJDK 11.0.2 on a Windows 10 machine with 16*GB* RAM and 4*x*3.2*GHz* Intel CPU.

Further, we reused the model from [FH16] and [Gru19] and made slight modifications and applied the required calibration. In the following, we describe the modification and the calibration of the memory hierarchy model:

Repository Model: Since we use the same example as in [FH16], we can reuse the repository diagram completely. The most relevant element in the repository model is the `MatrixMultiplicationComponent`, which provides the method multiplyMatrix. As we use the resourceCall, we additionally need to specify the `resourceCallRole` for the `MultiplyMatrixComponent`. We store the required resource for the call in the `MemoryHierarchyPlugin`, and we can access it via the pathmap mechanism. Figure 8.5 shows the model for the repository diagram.

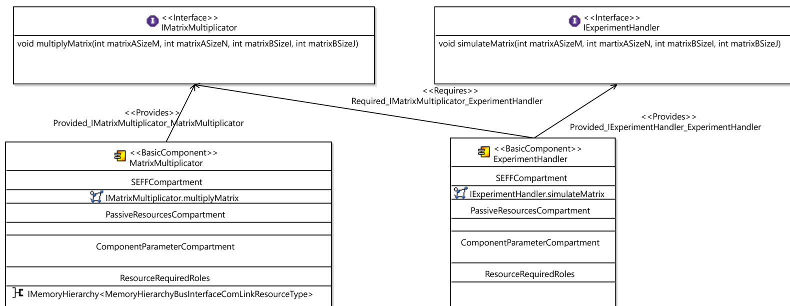


Figure 8.5.: Repository Model for the Matrix Multiplication Use Case

SEFF Model: Inside the SEFF diagram, we specify the actual behaviour of the multiplication. Since we use different hardware and thread numbers as in [FH16], we need to remodel this diagram—but keep the concept. We assume that the Pyjamas implementation of OpenMP splits the load equally on all threads. Thus, we use fork action, which contains 192 ForkBehaviours. Each behaviour includes a fraction of the actual load. Since the manual modelling of all 192 ForkBehaviours is time-intensive and error-prone, we can use the parallel loop AT from the parallel pattern catalogue (see Chapter 6). We use the measurements from the sequential run to calibrate the CPU demand. Thereby we separated CPU demands as good as possible from memory hierarchy demands. To achieve this, we also used the measurements we gain from perf (see Appendix A.5.2 for more information).

Additionally, we specify the resource call for the memory behaviour here and use the values provided by perf. Figure 8.6 shows the model for a two-threaded application using the fork action.

ResourceEnvironment Model: The modelling of the resource environment is straightforward and follows the example of [FH16]. However, we decided against using the exact schedulers from [Hap08] because for short response times, the exact scheduler implementation always adds a constant of $100ms$ to the simulation results. That can affect the simulation accuracy too much—especially for low prediction values.

Most important is that we add the stereotype for the memory hierarchy here.

MemoryHierarchy Model: The memory hierarchy model contains the new diagram type we included to model the memory hierarchy. Thus, we need to model it from the sketch. Figure 8.7 shows the final model.

We have to specify all attributes for the model elements identified in Section 8.2.1. We calibrate the values as follows:

Cache hit rate: To calculate the hit rate, we use the measurements from perf. Since the cache hit rate varies for each configuration of worker threads, we have to adjust the value for each experiment.

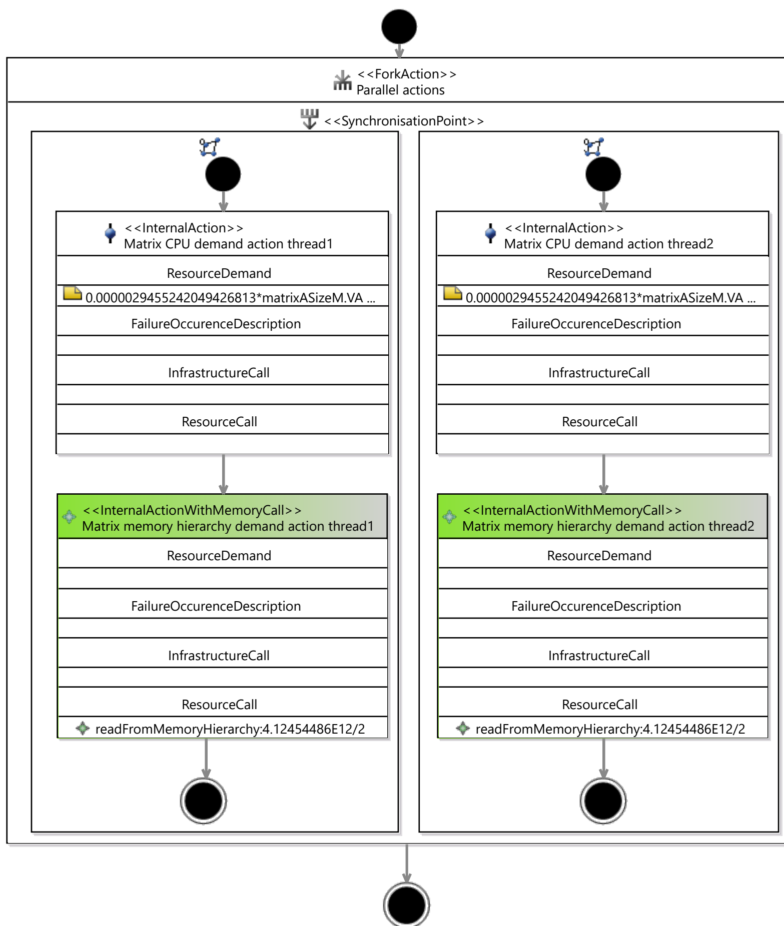


Figure 8.6.: SEFF Model for the Matrix Multiplication Use Case with Two Threads.

Cache isPrivate attribute: We establish whether a cache is private or shared from the CPU specification. In our case, L1 and L2 are private and L3 is shared.

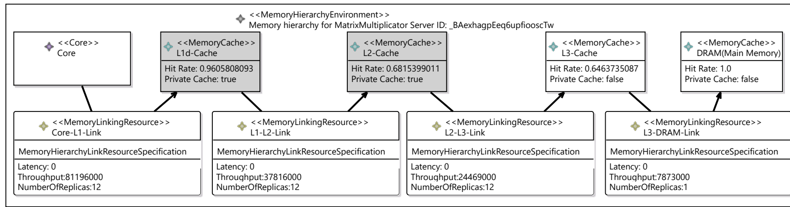


Figure 8.7.: Repository Model for the Matrix Multiplication Use Case

Memory link throughput: We use the measurements form memtest86 to get the memory link throughput for each hardware environment.

Memory link latency: We have not yet considered latency. Thus, we set this value to zero.

Memory link replicas: Set to the number of physical cores, since we assume that each core has its own memory bus. This value also represents the upper boundary, so hyper-threading introduces new virtual cores but no additional memory links.

With the calibrated model we described, we can perform the simulations. In the next section, we compare the simulation results and the measurements and discuss the outcome.

8.5.3. Results

To make the process of how we gain the results and the designs comprehensible, we discuss the outcome of the modelling and simulation for each iteration. In total, we have four iterations; iteration 0 describes the state before we include the memory model and iteration three a complex model with only a low accuracy increase. After the discussion of the four iterations and models, we make a comparison of all models in the next section. To better follow the result description for the individual iterations, we refer to the figures of the comparison in the next paragraph (see Figure 8.8).

8.5.3.1. Iteration 0: Default Palladio Model

Overview: The default Palladio model contains no memory attributes and represents our starting point. Modelling a parallel system (e.g., the matrix multiplication) follows the example in [FH16]. Here we use a fork action to specify the software behaviour of each OpenMP thread individually. To calibrate the CPU demand, we use the measurements to form the sequential run.

Model Modifications: None

Results:

12-Core System: The accuracy stays below an error of 20% for up to ten worker threads (for both the small and large use case). Afterwards, the prediction error continuously increases up to 55% for 24 worker threads. The predicted speedup is linear from one to 24.

40-Core System: The simulations predict a linear speedup as well. But the matrix multiplication scales on the 40-core system are worse than on the 12-core. Thus, the prediction accuracy is even worse and reaches an inaccuracy of more than 20% already when using four cores (small use case) or two cores (extensive use case). For 80 threads, the inaccuracy increases up to 65% for the extensive use case.

96-Core System: This effect becomes even more severe for the large system, which shows a maximum inaccuracy of almost 80% for 192 worker threads.

8.5.3.2. Iteration 1: Read-Data Model

Overview: The read-data model takes the amount of data required for the matrix multiplication into account. Further, it takes into account the different cache levels and the time needed to transfer the data from caches or main memory to the core.

Using this model, we have two options. First, we apply the memory hierarchy values to the model and do not cache the values for the CPU demand.

However, in the execution of the sequential run, we already consider a data transfer and cache hit rates—even if not explicitly. Thus, the second and more appropriate option is to also adjust the CPU demand by querying the data transfer demand.

Model Modifications:

Knowledge: Information about total data transferred.

ResourceCalls with memory demand in each ForkBehaviour and memory hierarchy model.

Memory Hierarchy: Setting of all attributes in the memory model diagram.

Results: For all systems and all use cases, the read-data model gives a more accurate prediction. However, the overall accuracy is only slightly better than the pure Palladio approach.

8.5.3.3. Iteration 2: Cache-Line Model

Overview: Because the accuracy of the read-data model is low, we further investigate a more refined grain memory model. In the cache-line model, we consider the fact that a cache miss will not only fetch the required data from the next memory level, but will also load a full cache line. In the above model, we assume a data transfer of $4bytes$ in case of a miss. In this model, we assume a transfer of $64bytes$ instead.

Model Modifications:

Knowledge: Pure CPU demand.

Internal Action: Recalibrated CPU demand.

MemorryHierarchyLink: Throughput of MemoryHierarchyLinks that transfer cache lines is divided by 16.

L3 Cache is set to private.

Results:

12-Core System: While the previous models overestimate the performance, the cache-line models underestimate the performance for both the small and large use case. However, the prediction error decreases greatly for the small use case: 5% off for 12 and 20% off for 24 worker threads. For the large use case, the error increases up to an inaccuracy of 60% for 24 worker threads.

40-Core System: On this machine, the cache-line model shows the best results. It still overestimates the performance, but is in all cases more accurately than the other models. For the small use case, it has a prediction error of 27% for both 40 and 80 worker threads. For the large use case, the prediction error is 28% (for 40 worker threads) and decreases to 11% for 80 threads.

96-Core System: Considering the 96-core system, the cache-line model behaves similarly to the read-data model. For lower thread numbers the read-data model is slightly more accurate, while for a number higher or equal to the core size the cache-line model is a bit better. In general, the cache-line models show an error of 53% for 96 worker threads and an error of 58% for 192 threads.

8.5.3.4. Iteration 3: Cache-Line-Scaling-DRAM Model

Overview: Beyond the cache-line model, we investigated further and included the scaling effects of the memory bus between L3 and main memory, too. The bandwidth scaling is dependent on the number of threads used. Therefore, we used the measurements taken by perf to calibrate the model further and adjusted the throughput of the memory link accordingly.

Model Modification: Throughput between L3 and DRAM is modified depending on the numbers of worker threads.

Results:

12-Core System: For the large use case, the cache-line-scaling DRAM model works very well with a prediction error of only 3% for 12 and 11% for 24 worker threads. However, for the smaller use case, the model is worse than the previous model and similar to the read-data model.

40- and 96-Core System: Also on these systems the model behaves worse than the cache-line model and similar to the read-data model for both the small and the large use case.

8.5.4. Result Summary

Server	Experiment Variation	Mean Prediction Error 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 8.1.: Mean Prediction Error for the Different Use Cases and Modelling Approaches

Figure 8.8 shows all the above models in direct comparison for the two use cases. The diagrams show the prediction error in percentage. The closer a value is to zero, the more accurate the predictions are. In addition to that, we provide more detailed diagrams and the speedup curve in Appendix A.5. For example, we provide models where we did not limit the memory links to the physical core size. Thus, we assumed that hyper-threading also increases the number of replicas for a memory link.

As we can see from Figure 8.8, different models behave best in different scenarios. Thus, there is not one model that beats all. However, we are interested in the prediction of a highly parallel application. So if we ignore low numbers of worker threads (e.g., lower than the number of physical

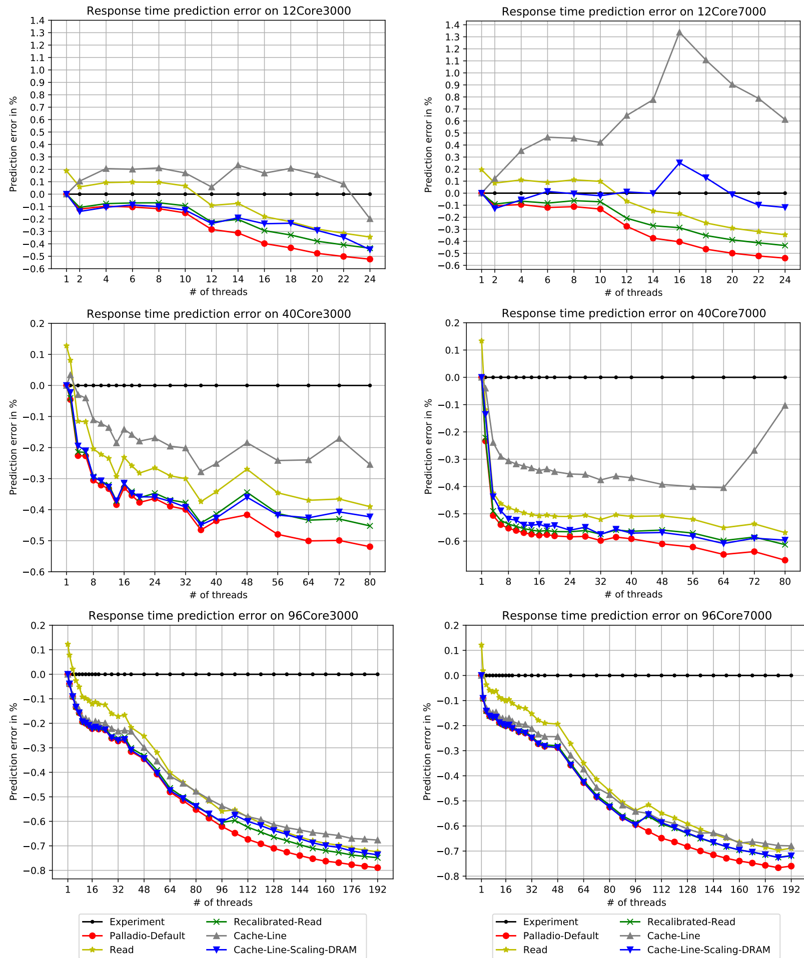


Figure 8.8.: Comparison of Prediction Models: Prediction Error in % for all Machines and Use Cases

cores), we can see that the cache-line model shows the best accuracy for all (except 12-core large use case) scenarios. For example, the cache-line model increases the accuracy for the maximum thread number and lies between 32% for the worst case (96-core system and small use case) and 89% for the best case (40-core system large use case).

Table 8.1 gives a full overview of the mean prediction error. The smaller the value, the more accurate the prediction. Bold values are the most accurate models for each row. We neglect the values for the read-data model because as we explained above, it uses a misleading calibration and considers memory effects twice. The mean prediction error is averaged for all thread variation. Thus, we cannot compare the error of the 96-core server with the 40-core server directly. That is because from 96 threads and up, measurements are only taken in steps of 8.

As we can see in the table for each scenario, we can find a model with a mean error below 40% and all models are more accurate than the default Palladio approach—except the cache-line model with the large use case. However, we are more interested in high values of worker threads. Considering the diagrams, we see that especially the accuracy of the 96-core system is inferior. This can have two reasons: (1) We did not capture all relevant characteristics in the memory model, or (2) there are other PPiFs that influence the performance. Given our current state of research, we believe the latter to be true. Especially effects such as data access, locks, and sequential parts of applications impact the parallelisation capabilities of highly parallel applications, and are not considered in the current models.

8.5.5. Discussion and Lessons Learned

During the creation of the meta-model extension and the experiment execution, we learned valuable insights that we want to share. Thus, we describe noteworthy lessons in the following.

Exact Scheduler: At first, we tried to use the exact scheduler developed by [Hap08]. However, we noticed that the implementations add an arbitrary but constant value of 100*ms* for short runtimes (below 200*ms*) to the predictions. This interferes with the result of the speedup for a large number of worker threads. On the other end, for very long

execution times, the exact scheduler seems not to add any demand at all.

Speedup: We used a small and extensive data set for the matrix multiplication because we assume the large data set puts more pressure on the memory architecture. Further, we expect to see this effect in the speedup. For the 12-core and 40-core systems, we did see the effect. However, on the 96-core system, the large data set showed a better speedup behaviour. That was surprising for us, and we can only assume that the cache architecture of the 96-core system is more effective than for the other two systems—also in prefetching necessary data.

Accuracy: Usually, the more cores we utilise the lower the response time is. However, this also means that a small absolute inaccuracy results in a sizeable relative inaccuracy. So, if we see fluctuations in the measurements (e.g., when garbage collection kicks in), we also see a temporary but significant fluctuation in the accuracy. Thus, we learned that the visualisation of the speedup curve is an excellent human-readable way to visualise the data and to detect errors manually.

DRAM Accesses: The main memory access is one of the most cost intensive operations. Thus, cache strategies try to avoid main memory accesses. As a consequence for us, it is most important to determine the number of main memory accesses as closely as possible.

System Utilisation: Some combinations of worker threads have a more positive effect on the performance than others. We assume the reasons lie in the NUMA nodes. Whenever a new NUMA node is used, more cache is available. On the other hand, for the data exchange, that means that data transfer to another NUMA node is more expensive.

Hyper-Threading: We researched the effect of hyper-threading on the memory models. We assume that virtual cores do not have private caches, nor do they increase the memory bus. Physical cores, on the other hand, have private caches and thus increase the overall cache size and memory bandwidth in the system. However, for the cache-line model, the separation of virtual and physical cores does not have an impact.

Cache-Line Model: The cache-line model works best for most cases. Especially on the 12-core and 40-core systems, the prediction results

are of good quality. Thus, it is all the more interesting that for the 96-core system, the prediction is so low. Obviously, we are missing performance-relevant factors. These factors might have something to do with memory bandwidth. For example, we did not research prefetchers, memory bandwidth latency or inter-core connections, which certainly have an impact on performance. However, it is more likely that they are of another nature and not bound to the memory hierarchy. Future work will have to look into that.

8.6. Threats to Validity & Limitations

To put the results in the right perspective, we discuss assumptions made and threats to validity in the following. Thereby, we distinguish between internal and external validity.

Internal Validity: Internal validity describes the validity of the specific experiment setting on which the response time prediction depends. Thus, we need to name three factors: the execution and measuring of the memory hierarchy utilisation, the experiment execution time, and the implementation of the simulation.

Multiple unforeseeable factors influence the execution time of the matrix multiplication. For example, we generate the matrix with random numbers. However, a matrix with many zeros can be calculated faster due to the internal processor optimisations. Also, we did not pin threads to cores but relied on the operating system's scheduler. So, threads could switch to other cores—which results in cold caches. Further, operating system interruptions can influence execution time. To minimise the effects, we executed each run multiple times and used mean values.

When taking the measurements, we increased the number of worker threads continuously. The execution of the experiments for all thread numbers would have resulted in very long execution times. Thus, we increased the thread number in steps, choosing a step size of four for numbers below 96 and a step size of 8 for numbers above 96. This is reflected in the calculation of the

mean error. Thus, we cannot directly compare the mean prediction error for the 96-core machine with the other machines.

Another threat to internal validity is the use of perf. Perf reads a low-level performance counter from the hardware to get, e.g., cache access rates. These performance counter events can vary between hardware vendors. For example, we are not able to read the `L1-dCache-store`. Even though the use of a performance counter was our only chance to get low-level information and is a common approach, the measurements might not be comparable between the hardware. Further, the use of monitoring applications puts additional overhead on the system, and can influence performance in general.

The next aspect we need to consider but have no influence on, is the TurboBoost or auto-throttling. Depending on the core's temperature, modern CPUs throttle down the CPU clock frequency. Thus, the CPU becomes slower. We did not investigate these effects, but monitored the CPU temperature during the experiments.

Finally, we have to discuss the model itself. During the meta-model creation process, we abstracted the architecture of the multicore CPUs significantly. Thereby, we neglected characteristics to make the model more comfortable to handle. However, it is possible to neglect performance-relevant aspects (e.g., prefetching or cache optimisation). One result of this might be the low accuracy of large multicore systems (e.g., a 96-core system). Following up on this, PPiFs is a task for future research.

External validity: The external validity describes whether the findings can be generalised outside the scope of this paper.

For now, we assume that the memory hierarchy model we developed can be generalised, because we analysed various CPU architectures upfront. The generalisation includes not only CPUs but also GPUs, even though we only focused on CPUs.

A more relevant threat is the evaluation scenario. In this work, we provide a proof-of-concept evaluation of the memory hierarchy model and the solvers. Thereby we used only one use case (i.e., the matrix multiplication), one programming language (i.e., Java), and one parallelisation paradigm (i.e.,

Pyjamas). However, a broader set of use cases, algorithms, languages, and complex applications is required to make more generalisable assumptions.

Finally, there are some minor threats, which go along with a controlled experiment. For example, we did not research how a system under load, with a complex application stack and multiple services running, will impact the memory architecture.

8.7. Summary of CB_3

In the course of this chapter, we focused on the requirements $R_{performance}$, $R_{modelling}$, and $R_{solvers}$ and researched an approach to consider memory hierarchies in performance predictions. To do so, we first identified PPiFs for memory hierarchies and their attributes. Next, we mapped the PPiFs to model elements and attributes, and included a memory hierarchy model in the PCM using a profile-based extension. Afterwards, we extended the editors to enable the SA to utilise the new model elements. Finally, we extended the current default simulator SimuLizar to interpret the added model elements and to take them into account during the simulations.

To evaluate the meta-model extension, we executed a matrix multiplication use case with different matrix sizes. At the same time, we modelled and simulated the use case and compared the measurements with the predictions. As a contribution of this chapter, we present:

1. Detailed insights into the memory behaviour of parallel applications.
2. A meta-model extension containing relevant model elements to model memory hierarchies.
3. A SimuLizar extension to interpret the model elements.
4. Four memory model approaches for different scenarios and with different prerequisites and levels of detail.

As a result of the contribution, we can show that the four memory model approaches increase the performance prediction accuracy of Palladio. Each model works exceptionally well under certain circumstances. Overall, we

favour the cache-line model, which has the best overall performance prediction power and increases the accuracy up to 57%. Thus, in the best case, the prediction error is below 11%.

However, the overall prediction for large systems and a high number of worker threads is still low with over 60% prediction error. We assume here the impact of further PPiFs, e.g., effects like data access, locks, and sequential parts of the application. To investigate these PPiFs is a challenge for future work.

To sum up, we can answer our research question:

$RQ_{3.3}$: **Can modelling the additional performance-influencing factors improve the overall accuracy of performance prediction?**

Answer: *We introduced a memory hierarchy model and included it for evaluation into the PCM. The results show that modelling the memory hierarchy helps in all cases to increase the performance predictions compared to the pure Palladio approach. For systems up to 40 cores, we even gained results that satisfied our requirements* $R_{accuracy}$.

9. CB_4: CPU Simulators

In this chapter, we introduce a different approach to tackle the requirements $R_{accuracy}$ and $R_{solvers}$ by using and integrating already available CPU simulators into the Palladio approach.

CPU simulators are often used by hardware vendors to benchmark their architectures [AS16]. CPU simulators have the advantage of reflecting the exact behaviour of specific CPUs, ranging from the CPU times up to the utilisation of the individual CPU registers. At the same time, this precise prediction of the behaviour comes at the cost of very long simulation times. Further, to utilise the simulators, we either need to provide a runnable application or the trace files of an execution.

Nevertheless, we are convinced that researching the integration of CPU simulators into the Palladio approach is beneficial, worth the effort, and can reveal new insights into the characteristics of parallel applications in multicore environments. Figure 9.1 shows the research approach and the structure of this chapter.

In the next section, we first explain the problem space, identify challenges, research questions, and set the goals. After that, we perform a structured literature search to identify available multicore CPU simulators, followed by an evaluation of all simulators. The assessment also includes the selection of suitable simulators. In the next section, we investigate extension strategies for Palladio. In combination with the selected CPU simulator, we prototype an extension process. Finally, we perform a use case evaluation and discuss the results and future work.

As a result, we provide a proof of concept approach, which we evaluate with the help of the bank account use case example (see Section 5.2.1). We are able to show that by using CPU simulators, the non-linear speedup behaviour is

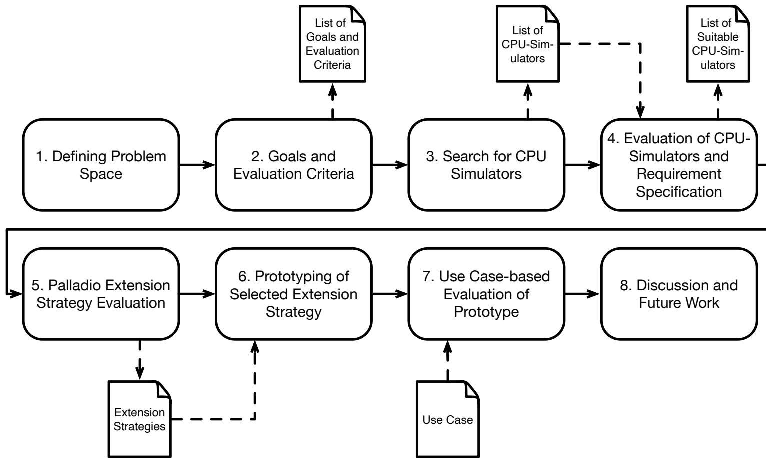


Figure 9.1.: Overview of the Research Method for Contribution CB_4

present in the performance predictions. However, the predictions underestimate the performance by far. This indicates that the input model is missing relevant characteristics.

Please note that significant contributions described in this chapter were part of collaborative student research projects [Det20; Gra18][1].

Additionally, we published all accompanying data (e.g., documentation on CPU simulator's docker files, implementations, evaluation data) online:
`https://zenodo.org/badge/latestdoi/282948837`

9.1. Problem Space

As we learned from the research in CB_1 to CB_3, predicting the behaviour of parallel applications is highly complex and depends on many PPiFs. In CB_3

[1]Please check them as well for further information, especially on implementation details.

we research the impact of PPiFs. Thereby, we looked only at one PPiFs at the time, knowing that the PPiFs influence each other.

Therefore, in the following, we research the integration of CPU simulators into the Palladio approach. CPU simulators predict the behaviour of an application on specific hardware in detail [AR06] and also consider side and cross effects of PPiFs.

9.1.1. Idea and Goal

To reflect the complex interaction of multiple PPiFs, we integrate existing exact multicore CPU simulators into the Palladio approach and utilise them as third-party model solvers.

To do so, we use Palladio's software, hardware, and usage models as input for the CPU simulators. Once we fetch the results from the simulators, we play them back into the Palladio Bench for further analysis.

In detail, we research and evaluate two different approaches:

Trace-based: We use SimuCom to extract the stack trace files. Next, we use the trace file as input for trace-based CPU simulators.

Source Code-based: We utilise ProtoCom to generate a Java jar file from the PCM. We use this jar file as input for source code-based CPU simulators.

9.1.2. Problem Specification

To successfully reach our research goal, we have to answer a set of questions:

RQ_{sim1} *Which CPU multicore simulators are available?* First, we need to know which CPU simulators exist and which purpose they serve. For this, we perform a structured literature search.

RQ_{sim2} *What are their advantages and disadvantages?* To be able to pick the appropriate CPU simulator, we not only need to know which ones exist, but also which advantages and disadvantages they have.

RQ_{sim3} *Is it possible to connect these simulators to Palladio and does the PCM provide enough data for the simulators?* To utilise the simulators from the Palladio Bench, we have to develop an integration strategy. Thereby we need to meet the requirements of the simulator.

RQ_{sim4} *If so, are the predictions more accurate?* Even if we can make use of CPU simulators, we have to make sure that the results we can gain from the simulators (based on the PCM input models) are more accurate than existing predictions.

To answer these research questions, we follow the research method illustrated and explained above (see Figure 9.1).

9.2. Overview of Multicore CPU Simulators

In this section, we give an overview of multicore CPU simulators. In a first step, we define the research strategy to find simulators in literature. Second, we give a short overview of all the simulators found, including their strengths and weaknesses. Finally, we present an overview, categorisation, and analysis of the simulator.

To follow the section, we recommend reading the section on CPU simulators in Chapter 2.4.1 first.

9.2.1. Search Strategy

To answer the research question RQ_{sim1}, we conduct a structured literature search. Since we assume the number of available CPU simulators to be low to moderate, we perform a simple keyword search using five databases (Google Scholar, IEEE explore, Research Gate, Science Direct and IBS BW). The keywords we use are *multicore, cpu and simulator*, which we combine into the single search term `multicore cpu simulator`.

In a second step, we perform snowballing to reveal additional simulators taken from related work.

We limit our result set to (a) multicore simulators, which are (b) not older than ten years (last update). Further, we have a set of requirements. So we are looking for CPU simulators, that:

1. can simulate Java applications
2. are suited for ISA x86, the most common architectures.
3. can be run under either Windows, MacOS, or Linux.

After conducting the search strategy, we sustained ten multicore CPU simulators.

Figure 9.2 gives an overview of the found simulators, categorising the simulators based on their capability to simulate Java applications and ISA x86 architectures.

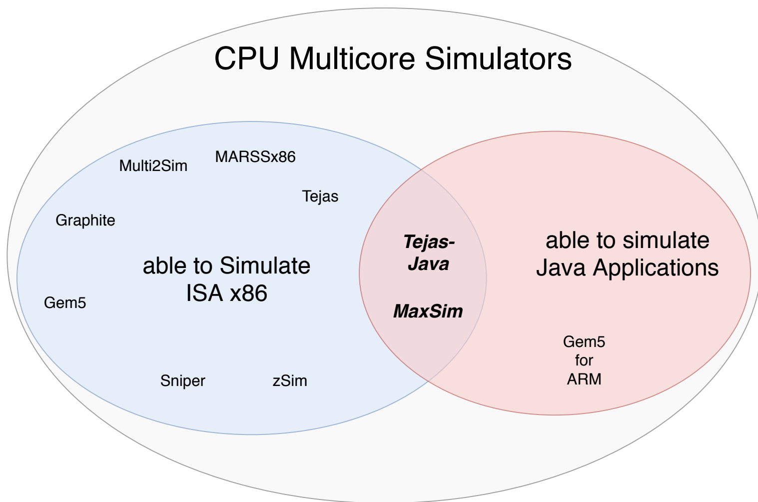


Figure 9.2.: Overview of multicore CPU simulators [Gra18]

In the following, we characterise all remaining simulators briefly. Thereby, we start with trace-based simulators and continue with source code-based simulators.

To characterise all simulators, we used available literature, set up, and ran example projects for all simulators. Thereby, we used Docker to handle dependencies and guarantee simple reuse. A description of how to run the Docker files is available in [Gra18] and all files are publicly available online[2].

9.2.2. Trace-based Simulators

We only found one CPU simulator that takes trace files as input.

Tejas: Tejas[3] is a multicore simulator designed by the Indian Institute Of Technology (IIT). It is entirely written in Java and was released in 2015 [SKK+15].

Figure 9.3 gives an overview of the main characteristics of Tejas. The dimensions of the spiderweb diagram are explained in detail in Section 2.4.1. The more a simulator fulfils a dimension, the closer the point is to the outer circle.

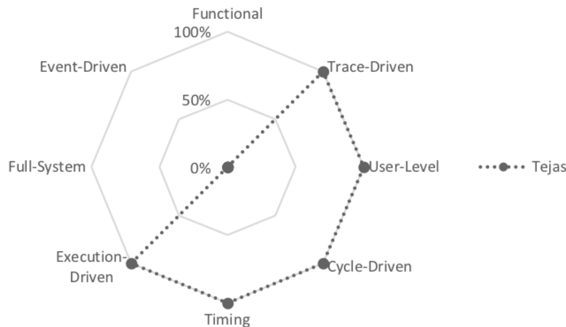


Figure 9.3.: Tejas Feature Net [Gra18]

The developer follows a cycle-accurate trace-driven approach. However, the core Tejas implementation requires two input files: first, the configuration

[2]https://doi.org/10.5281/zenodo.3961930
[3]`http://www.cse.iitd.ac.in/tejas/`

file; second, an executable file. This makes the core Tejas implementation a source code-based simulator. Further, like most other simulators, the Tejas approach uses the Intel PinTool. However, this tool only works with C++ code.

To support Java code, there exists an extension call Tejas Java[4]. Instead of the Intel PinTool, it uses the common *Jikes RVM*[5]. With the help of the Jikes RVM, it is possible to provide a trace file as input. Tejas Java can create stats and an output trace, which can be used as an input file for the original Tejas simulator.

9.2.3. Source Code-based Simulators

In the following, we briefly characterise the remaining CPU simulators. All of these are source code-based, and they need at least two input files: first, the simulator's configuration file and second, a compiled Executable and Linking Format (ELF) file.

Sniper: Sniper[6] is developed by a cooperation between the Ghent University and the Intel ExaScience Lab. Like most CPU simulators, it relies on the Intel Pin Tool and thus supports only C++ applications.

Figure 9.4 shows the characteristics of Sniper.

Sniper is a timing-based simulator, using a hybrid cycle simulation model. The hybrid model enables Sniper to skip specific cycles and gives a performance gain. Sniper is highly suited to simulate OpenMP applications.

zsim: Another CPU simulator has been developed by the Massachusetts Institute of Technology and Trustees of Standford University and further modified by MIT—zsim[7].

Figure 9.5 gives an overview of the characteristics of zsim.

[4] `http://www.cse.iitd.ac.in/tejas/tejas_java/`
[5] `https://www.jikesrvm.org/`
[6] `http://snipersim.org/`
[7] `https://github.com/s5z/zsim`

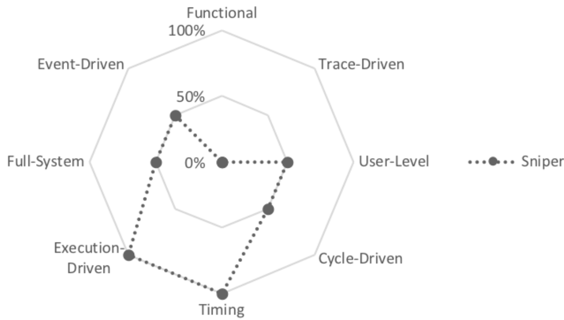


Figure 9.4.: Sniper Feature Net [Gra18]

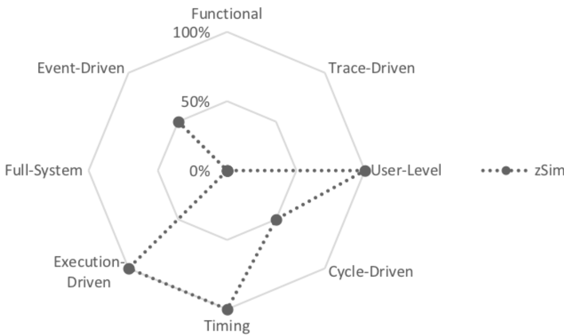


Figure 9.5.: zsim Feature Net [Gra18]

Zsim aims to simulate systems with up to 1,000 cores, and therefore they choose an execution-driven, user-level approach [SK13b]. zsim can simulate multi-thread and client server applications, and supports C++, Java, Scala and Python.

MaxSim: MaxSim[8] is a simulator built upon the Maxime VM and the zsim simulator. Therefore, the feature net looks similar (see Figure 9.6).

In contrast to most other simulators, MaxSim uses the Maxine VM instead of the Intel PinTool. This enables MaxSim to simulate Java applications as

[8] `https://github.com/beehive-lab/MaxSim`

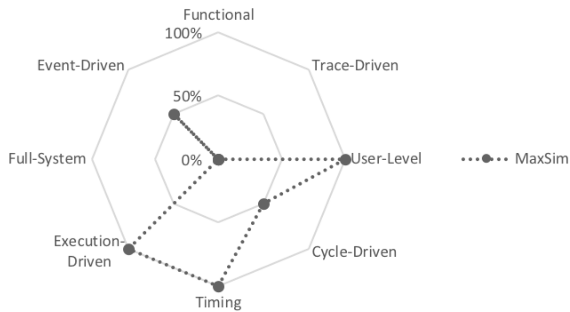


Figure 9.6.: MaxSim Feature Net [Gra18]

well. Further, the Maxine VM is capable of interpreting Java files newer than JDK 7.

Gem5: Gem5[9] is the fusion of the previous projects Michigan m5 and the Wisconsin GEMS. Scientists mainly use it for performance measurements and analysing computer architectures [BGOS12].

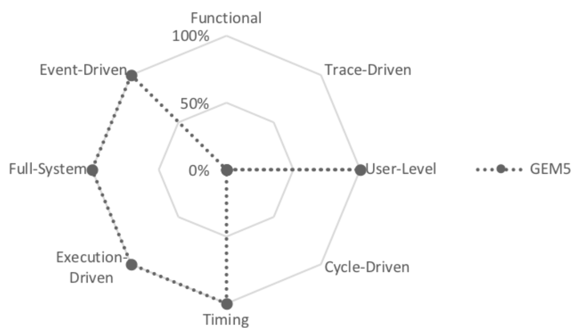


Figure 9.7.: Gem5 Feature Net [Gra18]

Figure 9.7 shows the feature net of Gem5 and indicates that Gem5 is an emulation-based simulator for x86 ISA architectures. Gem5 offers a set of

[9] `http://www.gem5.org/`

ARM ISA options, which gives much freedom. The emulation comes at the cost of performance and accuracy since Gem5 is not cycle-accurate.

However, Gem5 offers direct support of Java benchmarks.

MARSSx86: In contrast to Gem5, MARSSx86[10] is a cycle accurate full system simulator for x86 multicore ISAs.

The purpose of MARSSx86 is to have an efficient and straightforward complete system simulator [PAG11b]. Even though the full source code is available on GitHub, it is written in C code, and development ended in 2012.

Figure 9.8 shows the full feature net of the simulator.

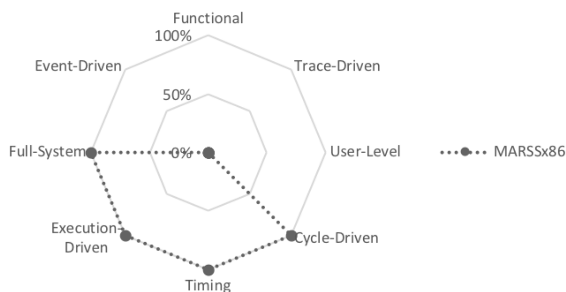


Figure 9.8.: MARSSx86 Feature Net [Gra18]

Multi2Sim: The purpose of Multi2Sim[11] is to support computer architects in the task of developing new architectures. Its primary goal is to verify the correctness and feasibility of new hardware designs [UJM+12].

Figure 9.9 shows the feature net of the simulator. It indicates that Multi2Sim is very versatile. Besides the capability of simulating x86 ISA, it can also simulate ARM and GPUs.

[10] http://marss86.org/
[11] http://www.multi2sim.org/

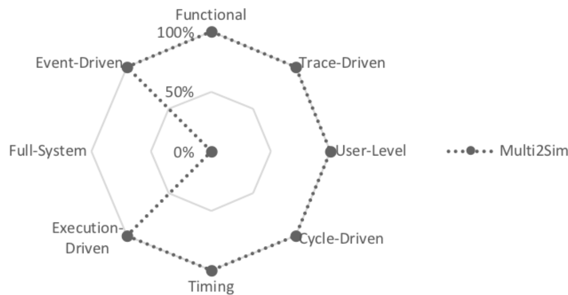


Figure 9.9.: MultiSim Feature Net [Gra18]

9.2.4. Evaluation and Selection

After conducting the search, we execute an assessment of the simulators. Thereby, we evaluate nine criteria. We are able to determine five of them by reading documentation or studying the corresponding literature. For the remaining four, we set up the simulators and use benchmark testing. In the following, we explain each criterion and how it is accessed.

ISA x86 Support: This is a yes or no attribute that describes whether the simulators support ISA x86, which are the most common architectures for desktop PCs nowadays. We get this information from the documentation

Coding Language: This attribute gives the programming language in which the simulator is written. We get this information from the documentation, literature, or by looking at the code repository.

Intel Pin Tool: This is a yes or no attribute. It describes whether the simulator uses the Intel Pin Tool. We get the information from the documentation

Input Type: This attribute describes the kind of input the simulator needs. We distinguish between trace file and runnable input. Further, we list the supported programming languages.

Processor Model: This attribute describes the supported processor models. We distinguish between in order (IO) and out-of-order (OOO). We gain this information from the documentation.

The following four characteristics cannot be extracted from literature, and are gained from setting up the simulator and running a benchmark example. All characteristics are raised by the execution of a single-use case, and therefore have limited power and are objectively biased. Nevertheless, we provided them as an indicator and an internal comparison.

Setup Difficulty: Describes how much work and time is needed to set up the environment and simulator until the first simulation result can be achieved. We also include time for fixing dependencies and running a hello world example. We measured the total time and gave the simulators with the highest time the attribute high, and the ones with the lowest the attribute low.

Community Support: Describes how active the community behind the simulator is and if they provide support. For the first, we looked at the dates of the last commit. For the latter, we sent a question to the community. If we received an answer within two weeks, we rate the community support high. Otherwise, low.

Configurability: This attribute describes how much freedom the simulator offers for configuration. We gain the information partly from the documentation, partly from testing.

Accuracy: Describes how accurate the predictions of the simulator are. To estimate the accuracy, we run each simulator with the following official benchmark SPLASH-2, PARSEC [BKL08], and SPEC CPU-2006[12]. Since these are common benchmarks, we can find results for some simulators and benchmarks already in the literature. In total, we use the values found in literature, and run the benchmarks multiple times on our own, calculating the average absolute accuracy error.

For all simulators and characteristics, Table 9.1 provides an overview. Given that overview of available multicore CPU simulators, we are able to answer RQ_{sim1}. Moreover, the overview of their advantages and disadvantages answers RQ_{sim2}.

[12] `https://www.spec.org/cpu2006/`

Given our description, evaluation, and testing, we nominate the simulator MaxSim for source code analysis and the simulator Tejas Java for trace file analysis, as promising candidates.

In the following, we sketch the process for including source code-based and trace file-based simulators into the PCM workflow.

9.3. Palladio Extension Strategies

In the last section, we provided an overview of all available multicore CPU simulators. We sketched their characteristics and briefly described advantages and disadvantages.

With this knowledge, we will design two strategies to include trace-driven and source code-driven CPU simulators into the Palladio approach. For both procedures, we will first theoretically describe how inclusion could work. Next, we provide a proof-of-concept evaluation with a CPU simulator most suited for the scenario. Finally, we will discuss the limitations of each strategy and further challenges to tackle.

9.4. Trace-driven Strategy

Since the Palladio models contain information on an abstract architectural level, the trace-driven inclusion strategy sounds most promising. The general idea follows the concept to extract the stack traces from one of the Palladio's solver engines. In the next step, we use the traces as input files for the CPU simulators and run the simulations. Finally, we play back the results from the simulators to the solver.

Figure 9.10 exemplifies this process. As shown, we do not use any additional information besides the already existing Palladio models. As a solver engine, we propose SimuCom, because SimuCom uses m2t transformations to generate simulation code, which provides resource demand traces. In the following, we have a detailed look at the SimuCom solver and the ways to extract traces.

	Multicore CPU Simulators							
Characteristic	**Sniper**	**Gem5**	**MaxSim**	**MARSSx86**	**ZSIM**	**Tejas**	**Tejas Java**	**Multi2Sim**
Multicore Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ISA x86 Support	Yes	(Yes)	Yes	Yes	Yes	Yes	Yes	Yes
Code Language	C++	C++	C++	C++	C++	Java	Java	C++
Intel PIN used	Yes	No	Yes	No	Yes	Yes	No	No
Input Type	C++	C++, (Java)	Java	C++	(Java), C++, Python	trace_file, C++	Java	C++
Processor Models	IO, OOO	IO, OOO	IO, OOO	IO, OOO	IO, OOO	IO, OOO	Io, OOO	OOO
Setup Difficulty	hard	easy	easy	–	hard	hard	hard	easy
Configurability	high	very high	high	low	high	high	high	high
Community Support	high	low	low	low	low	high	high	high
Avg. Accuracy Error	23.8%	9.7%	– –	50%	11.2%	18.8%	18.8%	17.5%

Table 9.1.: Overview and Compression of CPU Multicore Simulators [Gra18]

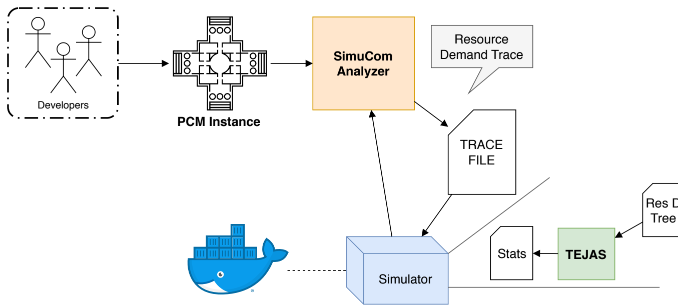


Figure 9.10.: Inclusion Strategy Using SimuCom and Trace-driven Multicore CPU Simulators [Gra18]

9.4.1. SimuCom

As explained in Section 2.4.2.4, the SimuCom approach follows a m2t transformation approach to generate simulation code out of PCM instances. The SimuCom framework uses and executes the simulation code to simulate the system.

As part of the suitability check and analysis of SimuCom, we identified two possible extension points in the source code of the SimuCom Framework (see Appendix A.6.1).

The first extension point (see Listing A.7 in Appendix A.6.1) hooks into the `getScheduledResource`-method. At this point, the processed-demand traces are available.

The second extension point (see Listing A.8) hooks into the `ExperimentRunner`. At this point, the stochastic simulation starts. Here, the idea is to get the event traces and hand them over to the CPU simulator.

9.4.2. Discussion

Unfortunately, we are not able to implement the trace-driven approach without immense effort and without changing either SimuCom or Tejas

significantly. The main reason for this is that most multicore CPU simulators do not accept trace files. The only exception is the Tejas simulator.

However, the trace files provided by SimuCom are not suitable for analysis with Tejas, because they lack more detailed information about software and hardware. SimuCom only provides resource-demand traces, but Tejas needs additional information about the CPU architecture, memory addresses, and operations.

In a nutshell, we believe that the trace-driven approach is still worthy of future research. However, CPU simulators are designed to help CPU designers evaluate the design of a CPU architecture, and therefore require much low-level information and return very detailed information about the status and behaviour of the CPU. For our purposes, this information is too detailed, and, at the same time, we are not able to provide the amount of input data required, since we use Palladio to look at architectural design.

So, for future research, we propose having a look at high-level multicore thread simulators, if available, or extending the current state-of-the-art Palladio simulator, SimuLizar. Thereby, we can use the insights of the CPU simulators and also use their libraries like JIKES or Maxine VM.

9.5. Source Code-Driven Strategy

Realising that the trace-driven approach does not work out of the box, we have a closer look at the source code-based approach. Figure 9.11 lays out the source code-based approach. As in the trace-driven approach, we use a PCM instance as a starting point. This time, however, we do not use a simulator to generate the trace, but use ProtoCom to create a runnable Java SE performance prototype.

We feed the performance prototype to the CPU simulator and play the results back to the Palladio Bench. In Figure 9.11, we show the removal of all Java RMI calls and other overhead. This step is required, since most CPU simulators cannot handle RMI calls well.

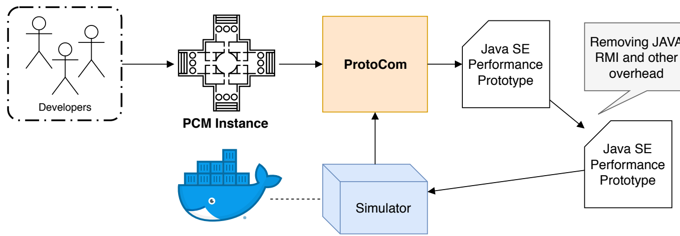


Figure 9.11.: Inclusion Strategy Using ProtoCom and Source Code-base Multicore CPU Simulators [Gra18]

9.5.1. Removal of Java RMI Communication

Removing the Java RMI communication from the ProtoCom performance prototype meant a manual adaptation of the generated source code and the elimination of all the features coming with Java RMI calls (e.g., the simulation of distributed systems).

However, this step is necessary, because all the remaining CPU simulators (which support Java files) were not able to successfully run RMI calls. The underlying engines Jikes RVM or Maxine VM do not support Java RMI calls, and even with the help of the engine developers, we were not able to include this feature in a reasonable amount of time. Thus we have to remove all RMI calls to proceed.

To still be able to run the prototype, we unravel the RMI communication stack trace and include a new class calling the required methods not via method invocation, but by simply calling the required method in the specific order (for further implementation details, please see Appendix A.6.2).

The removal of the RMI calls is only possible due to our simple use case, and would result in a complex task for larger, distributed, or more advanced use cases.

9.5.2. ProtoCom Calibration

To be able to run the simulations locally but still get the correct results for the target system, we first need to calibrate the ProtoCom performance prototype.

Therefore, we execute a calibration run, which includes the simulation of a Java prototype. The prototype performs a fixed number of, e.g., Fibonacci demand operations on the target system. We use the measurements to create a calibration table. With the help of the calibration table, we are now able to execute the simulations locally, while getting the correct results for the target system.

9.5.3. Discussion

In the above sections, we have sketched a method to use a PCM instance as input for ProtoCom, generate a runnable performance prototype, and use the prototype as input feed for multicore CPU simulators.

However, this process is not straightforward, contains a lot of manual adaptations, and only works for specific use cases. One of the major drawbacks is the lack of support for Java RMI calls by the CPU simulator's engine. Further, the benefit of the simulator itself is questionable, since only two simulators (MaxSim/zsim and Tejas) support Java applications at all, and their accuracy (11.2% and 18.77%) is medium for real applications, which means we can assume that the accuracy drops even further when generating performance prototypes out of abstract architectural models.

Nevertheless, we were able to sketch the process of including CPU simulators into the Palladio process, and therefore successfully answered RQ_{sim3}.

9.6. Execution and Use Case Evaluation

To answer the final research question RQ_{sim4}, we perform a use case evaluation of the source code-based approach using the multicore simulator

MaxSim. All results, configuration files, Docker containers, and measurements are publicly available[13].

9.6.1. Use Case and Process

As a use case, we use a complex example of the bank account use case (see Section 5.2.1). We decide to use this example for multiple resources:

1. We performed a performance prediction of this use case and received a very poor accuracy of 63% for 16 cores in [FSH17].
2. We assume the reasons for this poor accuracy lie in the complex interaction of different PPiFs, which CPU simulators are supposed to handle well.
3. With an error of 11.2%, the use of MaxSim should significantly improve the accuracy of the predictions.

The process we follow to evaluate the CPU Simulator approach is straightforward. We use the measurements taken in [FSH17] as ground truths. This gives us (a) the measurements from implementation and execution of the use case, (b) the results of the Palladio simulation, without any extensions, and (c) the PCM models for the use case.

In the next step, we use the PCM models to generate the performance prototype using ProtoCom. Next, we adopt the prototype as described above, remove all RMI calls, and perform the ProtoCom calibration process to run the simulations for the target system locally. After that, we feed the prototype to the MaxSim simulator, and finally, we compare the results from the simulator to the measurements and Palladio simulation results from [FSH17].

9.6.2. Setup

The setup phase contains two actions: the setup of the simulator and the calibration of ProtoCom.

[13] `https://zenodo.org/badge/latestdoi/282948837`

9.6.2.1. MaxSim Setup

During the setup, we have to configure the CPU simulator. The configuration includes specifying the characteristics of the CPU architecture. The listing in Appendix A.9 shows the full configuration file for MaxSim. It includes the specification of the L1, L2, and L3, as well as the specification of the number of cores and clock rates.

9.6.2.2. ProtoCom Calibration

To calibrate the local ProtoCom instance for the target system, we created a sample calibration project with a synthetic ProtoCom resource demand (e.g., calculating Fibonacci numbers). In this project, we specified the number of calculation iterations to $1,000,000,000$ and executed the project on the target system. The execution takes around $25.7s$ (see Appendix A.6.4 for more detailed information).

With this information, we can adjust the ProtoComs calibration table and include the information into the performance prototype.

9.6.3. Execution & Measurements

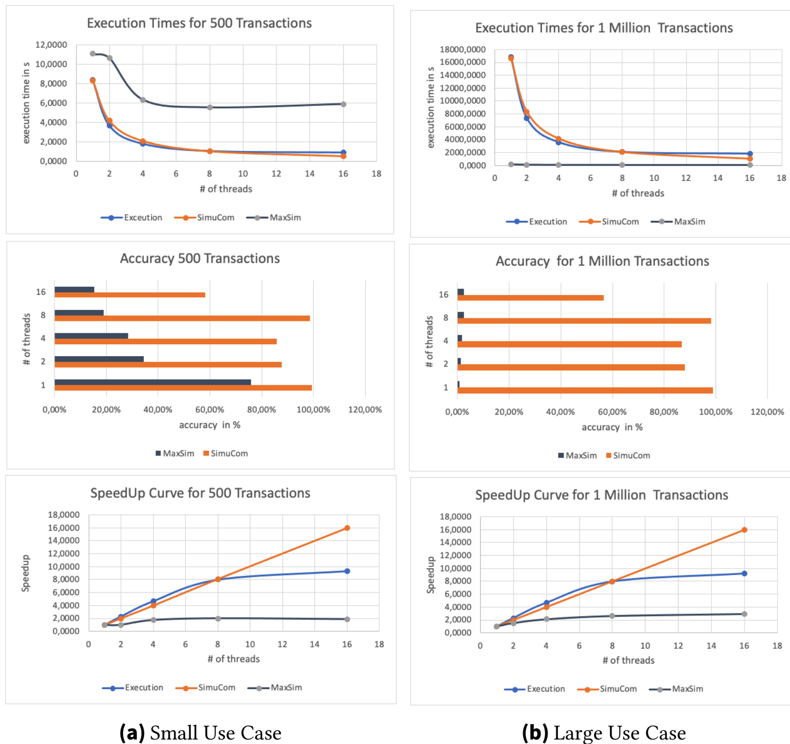
Due to a version change in Palladio and Java, we re-executed the simulations with Palladio using the same values and experiment setup as in [FSH17]. We get the same results and continue with the execution. Unfortunately, we are not able to re-run the experiments on the hardware, since it is not available any more. Therefore, we have to rely on our previous measurements.

Table 9.2 shows the measurements from [FSH17], the simulation results using SimuCom, and the simulation results using MaxSim for one to sixteen worker threads. The upper part of the table contains the result for 500 transactions (small use case) and the lower part the results for one million transactions (large use case).

Further, Figure 9.12 visualises the results using bar and scatter charts.

500 Transactions								
	Execution		SimuCom			MaxSim		
# Threads	**Time** in s	**Speedup**	**Prediction** in s	**Speedup**	**Accuracy**	**Prediction** in s	**Speedup**	**Accuracy**
1	8.40	1.00	8.35	1.00	99.42%	11.08	1.00	75.82%
2	3.66	2.30	4.17	2.00	87.69%	10.67	1.04	34.31%
4	1.80	4.67	2.10	3.98	85.78%	6.34	1.75	28.38%
8	1.05	8.00	1.04	8.07	98.57%	5.54	2.00	18.95%
16	0.90	9.33	0.52	15.96	58.14%	5.89	1.88	15.29%
1 Million Transactions								
	Execution		SimuCom			MaxSim		
# Threads	**Time** in s	**Speedup**	**Prediction** in s	**Speedup**	**Accuracy**	**Prediction** in s	**Speedup**	**Accuracy**
1	16807.09	1.00	16607.61	1.00	98.81%	135.06	1.00	0.80%
2	7316.18	2.30	8305.96	2.00	88.08%	88.93	1.52	1.22%
4	3606.82	4.66	4153.26	4.00	86.84%	63.71	2.12	1.77%
8	2117.25	7.94	2076.40	8.00	98.07%	51.33	2.63	2.42%
16	1833.90	9.16	1038.60	15.99	56.63%	45.78	2.95	2.50%

Table 9.2.: Overview of all measurements and simulation results from SimuCom and the MaxSim approach



(a) Small Use Case **(b)** Large Use Case

Figure 9.12.: Chart based visualisation of the measurements for small and large use cases

9.6.4. Discussion

Given the results of the experiment, the first thing to notice is the overall poor accuracy of MaxSim. Even for sequential scenarios, the results are very inaccurate. Overall, MaxSim performs a lot better for the small use case (accuracy up to 76%), but performs very poorly for the large use case (accuracy up to 2.50%). Hence, we can answer the question RQ_{sim4} and are not able to provide more accurate results with the use of CPU simulators.

Second, we noticed as—pointed out in [FSH17]—the super-linear speedup of real execution for two and four worker threads.

Third, when looking at the speedup behaviour (cmp. Figure 9.12) we can see that the CPU simulator captures the behaviour of the real application, but is off by a factor of 10 to 20. In contrast, we see that SimuCom applies a linear speedup.

To wrap it up, CPU simulators are used to benchmark a CPU architecture design. To do so, they give very detailed information on the behaviour and characteristics of a CPU. In the past, they were able to show that they work with high accuracy. However, to work properly, they need detailed information and runnable source code. So we assume that the reasons we got such inaccurate results are the following:

Missing Model Information: Palladio is used to analysing software architectures. Therefore, they rely on an abstract design of software, hardware, and user behaviour. These models abstract a lot of detailed information. The absence of this information can highly influence the performance prototype and therefore, the multicore CPU simulator. Besides, we do not model any multicore specifics in Palladio.

Simplified Model: To be able to create architectural models in Palladio, it is often necessary to abstract. Even if we have a correct source code implementation, we are going to lose information in the process of model creation.

Use Case: We chose the bank account use case, as it seemed suited for this approach, even though we pointed out the challenges in modelling this use case in Palladio and running it using ACTORS or OpenMP in [FH16; FSH17]. In retrospect, the selection of another use case or a parallel performance benchmark would have been more appropriate.

Measurements: Since the hardware we used in the beginning became inaccessible, we had to trust the previous measurements, and could not validate them. From the time the measurements were taken until the simulations were carried out, dependencies and the Java version changed, which could have influenced the outcome.

Incomplete CPU Model: CPU simulators are based on complex CPU models. For each CPU, we had to create the model by ourselves, due to the

absence of preconfigured models. This process is time-consuming and prone to errors. Small changes can lead to significant changes in the simulation results. To avoid errors, we used the information provided by the CPU vendors and tried our best to create accurate CPU models.

Artificial Load: The CPU load generated by ProtoCom is artificial. ProtoCom supports five different types. In our case, we used the default setting and created the performance prototype with a Fibonacci demand. Each demand has specific characteristics (processor intensive vs. I/O intensive). However, for the complex use case, a single demand type might be not sufficient.

9.7. Summary of CB_4

In this chapter, we discussed the possibilities to integrate multicore CPU simulators, used by hardware engineers, into the Palladio approach. To do so, we first executed a structured literature review to find the current state of the art in CPU simulators. Next, we evaluated each CPU simulator, carved out its strengths and weaknesses, presented the results in an overview table (see Section 9.1), and showed how they can be used by SA for performance predictions. In a second step, we sketched out the integration process of both trace-driven and source code-driven CPU simulators into the Palladio workflow.

Finally, we implemented and executed the source code-driven approach by using the CPU simulator MaxSim. Unfortunately, the results we received were very inaccurate and performed on average even worse than before. In the above section, we discussed the reasons for the inaccuracy. Two reasons we think have the most substantial influence are (a) the example use case used, and (b) the abstract input model.

When continuing the research, we first need to evaluate the results by the use of a second scenario. Further, we will try to use another CPU simulator based on another engine (Jikes RVM vs. Maxine VM). However, there is an even more significant challenge to face. All of the CPU simulators we tested cannot handle any Java files built with Java 1.8 or above. This technical

limitation and the fact that the CPU simulator engine cannot simulate Java RMI calls makes it close to impossible to continue research at the moment.

In conclusion, we answer our research questions (see Chapter 3) as follows:

$RQ_{4.1}$: **Can CPU Simulators be used by software architects to evaluate the response time of parallel architectural designs?**

Answer: *We were able to show that it is possible to transform the architectural models into a performance prototype, which we again can use as input for multicore CPU simulators to determine the response or execution time of a parallel application.*

$RQ_{4.2}$: **How would the integration of CPU simulators alter the process of performance predictions?**

Answer: *In Section 9.3 we sketched two approaches to include CPU simulators into the performance prediction workflow: (1) a trace-driven approach, (2) a source code-driven approach. In both cases, we use the PCM without additional information as a starting point. Next, we transform the PCM by the use of solvers either into a tracefile or a performance prototype, which we finally use as input for the multicore simulators.*

$RQ_{4.3}$: **Does the use of CPU Simulators increase the performance prediction accuracy for parallel applications in multicore environments?**

Answer: *We implemented the source code-driven approach to evaluate the accuracy of the performance prediction using multicore CPU simulators. Thereby, we used a complex use case example, the Bank Transaction Example (see Section 5.2.1). The prediction accuracy of this approach for the given example was very inaccurate, with an accuracy from* 2.50% *to* 15.29%*, and up to* 54% *worse than the pure Palladio approach.*

Therefore, we have to reject our hypothesis H_4*: CPU simulators—used in other domains (e.g., hardware vendors)—can help to improve the predictions for parallel applications on multicore CPUs.*

Evaluation & Summary

10. Evaluation

In the previous four chapters, we presented in detail the four contributions of this thesis. Along with a detailed description, we provided an extensive discussion about the benefits and limitations, and evaluated each contribution individually. In this chapter, we pick up our overall research goal (see Chapter 3), show how the contributions can be combined, give an overview of the research questions we answered, and show the contribution of this work given the requirements from Chapter 1.

10.1. Combination of Contributions

Even though we previously considered each contribution individually, a combination of the contributions is possible and even desirable. Thus, we will discuss whether and how a combination is possible.

10.1.1. Combination of CB_1

In CB_1 (Chapter 6) we researched the capabilities of the PCM language to express parallel behaviour. As a result, we provide a lightweight meta-model extension using the AT method and provide a pattern catalogue to quickly include common parallel patterns into the software models. The main characteristic of the lightweight extension is that we do not alter the core meta-model, and can map all new language elements to already existing ones. Thus, we ensure that all existing simulators and extensions can still handle the models. Further, this makes it theoretically possible to combine the parallel architectural pattern catalogue with all the other contributions.

In the following, we briefly sketch what a combination would look like.

Combination with CB_2 In CB_2 (Chapter 7) we researched the behaviour of parallel applications and the influence of PPiFs on performance. Further, we extracted performance curves to capture the characteristics of different types of resource demands. We included the performance curves into Palladio to enable the SA to increase the performance prediction without modelling the characteristics of parallel applications in detail.

In Section 7.6, we show how we integrated the performance curves into Palladio using the parallel pattern catalogue. Thus, this indicates that the combination of the two contributions is not only easily possible, but is even necessary in order to use the performance curves in Palladio.

Combination with CB_3 In CB_3 (Chapter 8) we extended the PCM to include memory architectures of CPUs into the PCM. Thereby, we extended the software and hardware models, as well as the simulator SimuLizar.

For a combination of CB_1 and CB_3, we have to have a detailed look at the SEFF diagram: To consider memory accesses, we altered the internal action element so that we can specify the memory access needed. To successfully use the pattern catalogue in combination, we have to ensure that during the QVT-o transformation (1) the internal action is copied with all attributes, and (2) the memory access demand is adjusted for each copied instance. Currently, the first requirement is fulfilled. The latter has not yet been implemented. However, an adaptation is easily possible, if we assume that the total memory access demand is spread equally amongst all threads, spawned by the parallel AT.

Combination with CB_4 In CB_4, we present a prototype approach to use a multicore CPU simulator as a solver for the PCM. Even though we achieved predictions of low accuracy with CPU simulators, a combination of CB_1 and CB_4 is possible without further actions.

As described in the Chapter 6 (CB_1), we ensure that all solvers still work due to the lightweight meta-models extension. Therefore, we can use both sketched strategies (trace-based and source code-based) in combination with the parallel pattern catalogue. Since the pattern catalogue focuses on the software models, using the extension will lead to faster creation of the models, but will not affect accuracy.

10.1.2. Combination of CB_2

We can use the developed performance curves to adjust the performance predictions, e.g., by adding additional resource demands to the model or by calculating the difference to the linear speedup. To gain the performance curves, we performed extensive experiments and used the measurements to extract performance curves using linear regression. Thus, the performance curves include a lot of implicit effects going on during parallel execution.

Given that, we have a look at the combination of the remaining two contributions and discuss whether a combination makes sense.

Combination with CB_3 While extracting the performance curves, we looked at various attributes: Number of worker threads, number of physical and virtual cores, performance (i.e., speedup), and the type of resource demand. While using the measurements from the experiments to extract the performance curves, we captured effects implicitly, such as synchronisation, caching, or idling. Thus, a combination of the performance curves with the memory bandwidth model is, in theory, possible.

In Section 8.5, we conclude that the cache-line memory model is the most fitting one. In the following, we briefly describe the results when combining the cache-line model with the performance curves. Thereby we use the matrix multiplication example as a reference use case. To gain the combined values, we first simulate the cache-line model as described in Chapter 8. Afterwards, we apply the performance curves manually, as described in Section 7.5.5.

Figure 10.1 shows the prediction error when combining the cache-line model with the matrix multiplication performance curve. Further, the figure shows the error for the different hardware and use case settings.

In addition to that, the following Table 10.1 shows the mean prediction error.

Looking at the pure values shows that the combined model works for the 40-core system and the 96-core system. However, it brings an accuracy decrease for the 12-core system. The interpretation of this observation is as follows: the performance curve always assumes an additional overhead. In the case

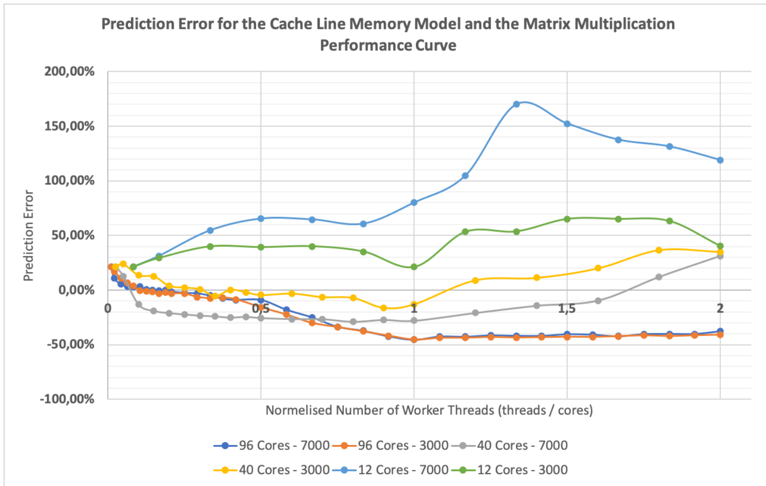


Figure 10.1.: Prediction Error for the Combined Approach: Matrix Multiplication Performance Curves and Cache-line Memory Model

Server	Experiment Variation	Mean Prediction Error Cache-Line [%]	Mean Prediction Error Cache-Line + Perf Curve [%]	Improvement [%]
12-Core	3000x3000	15.30	43.53	-28.23
	7000x7000	61.40	91.78	-30.38
40-Core	3000x3000	15.80	11.83	3.97
	7000x7000	29.80	21.82	7.98
96-Core	3000x3000	37.90	24.14	13.76
	7000x7000	37.50	22.34	15.16

Table 10.1.: Comparision of Cache-Line and Cache-Line with Performance Curvees

of the 12-core system, the cache-line model was already underestimating performance. Thus, by adding the performance curve, we increased the underestimation and made the predictions worse. For the other two cases, the opposite is true. The cache-line model overestimated system performance

under test. So, by adding the performance curves, we added additional overhead and increased the accuracy of the prediction.

In general, we do not suggest combining the memory model with the performance curve. The main reason for this is that while taking the measurements for the performance curves, we measured memory effects as well—even though the measuring was implicit by measuring the overall performance. Thus, both models, the performance curves and the memory model, include memory effects. Combining them would mean taking this effect into account twice. The increase in accuracy of the larger systems was only a lucky coincidence resulting from adding to inaccurate prediction approaches.

Instead of a combination of the two approaches, we suggest investigating the effects of PPiFs more in-depth, and making either approach more accurate.

Combination with CB_4 More interesting is a combination of the performance curves with the CPU simulator approach. Even though the performance curves do include most characteristics we want the CPU simulators to evaluate, we learned that our current input models are too abstract for the multicore CPU simulators to provide accurate results. Here the performance curves can give a boost. Using the performance curves with the parallel architectural pattern catalogue will result in adding additional overhead as internal action to the model.

Evaluating whether these models will result in more accurate predictions using the multicore CPU simulators is an open task and remains for future work.

10.1.3. Combination of CB_3 and CB_4

The remaining combination is the combination of the PCM extension for memory hierarchies and the use of multicore CPU simulators.

Unfortunately, a combination is currently not possible, because neither the SimCom solver (used for the trace-driven approach) nor ProtoCom (for the source code-based approach) supports interpretation of the memory hierarchy extension. Thus, there is no method to feed the memory models into the CPU simulators.

	CB1	CB2	CB3	CB4
CB1	-	✓	(✓)	✓
CB2	✓	-	✓	✓
CB3	(✓)	✓	-	✗
CB4	✓	✓	✗	-

✓ – is suitable
(✓) – needs minor adaptation
✗ – not supported

Table 10.2.: Summary of working combinations

Nevertheless, researching performance prototypes such as those created by ProtoCom, which includes the information from the memory hierarchy model and therefore memory accesses and cache behaviour, sounds promising and is an open challenge for future work.

To summarise the possible combinations, Table 10.2 gives an overview of which combinations are suitable.

10.2. Research Goal Evaluation

In the introduction (see Chapter 1), we motivated the problem for performance prediction arising from multicore CPUs and highly parallel software. We identified five requirements that we need to fulfil to enable accurate performance predictions for parallel applications in multicore environments. In Chapter 3, we defined the following research goal of this thesis:

> **Research Goal (*RG*):** Improving the accuracy, usability, and applicability of model-based QoS predictions concerning the performance of parallel applications in multicore environments.

Next, we refined the requirements given the RG and raised four research questions.

In this section, we evaluate whether we achieved each research goal. Therefore, we will first answer the research question, discuss whether the requirements were satisfied, and finally, assess whether the RG was achieved.

10.2.1. Answering the Research Questions

Because the research questions map to the contributions, each research question has already been discussed in the corresponding chapter. Therefore, we will not discuss them here again. However, in Appendix A.7, we provide a condensed version of the questions and our answers.

10.2.2. Assess Requirement Fulfilment

After going through the research questions and their answers, we revisit the following requirements we initially set up. In this step, we show which contribution did fulfil the requirements. Also, we lay out open tasks and challenges for future work.

10.2.2.1. Assess $R_{modelling}$

> $R_{modelling}$: *Software architects shall be able to express concurrency in software models, which describe software behaviour. This includes highly concurrent software, which can consist of multiple hundreds or even thousands of concurrently executed threads.*

With the help of the parallel architectural template catalogue (see Chapter 6), we provide an easy-to-use approach for the SA to quickly include massive parallel behaviour. Thereby, the parallel AT catalogue includes four abstract design patterns. The SA can use the four patterns to model the behaviour of 32 out of 35 common parallelisation patterns we identified in a structured literature review.

Further, we introduced a PCM extension to enable the SA to specify the memory accesses and memory data consumption (see Chapter 8).

Open Tasks: The remaining three patterns are based on message passing, which we have not yet considered. Therefore, two open tasks are: (1) include message-passing concepts (e.g., MPI or Actors); (2) include inter-thread communication. When designing the pattern catalogue, we focused on the specification of the thread behaviour. Up to now, we have not included inter-thread communication, which can influence the software behaviour, e.g., due to waiting conditions.

10.2.2.2. Assess $R_{metrics}$ and $R_{performance}$

> $R_{metrics}$: *In case the single metric—CPU speed—is no longer sufficient to cover all the performance relevant aspects for multicore systems, the software architect shall be able to specify additional performance-influencing factors (e.g., memory bandwidth, cache behaviour, or the memory architecture) needed.*
>
> $R_{performance}$: *The performance prediction models shall include relevant performance-influencing factors and reflect the additional complexity.*

To tackle this requirement, we provide two solutions strategies. First, we extended the PCM to include the memory architecture (see Chapter 8). That way the SA is now able to specify the L1, L2, L3, main memory, and memory bandwidth in the hardware model. Further, he can define the memory accesses and memory consumption in the software model.

The second strategies are to use performance curves. The performance curves we extracted from extensive experimentation (see Chapter 7) include additional PPiFs in an abstract way. The SA can use one out of six pre-defined performance curves to consider additional PPiFs in the performance predictions.

Open Task: As shown in Chapter 8, considering memory architectures in the performance predictions already helps improve accuracy. However, to be even more precise, we need to consider additional metrics. So, open for future work is investigating the PPiFs that have not yet been considered, and stepwise including the most relevant ones.

10.2.2.3. Assess $R_{solvers}$

> *$R_{solvers}$: The solvers, used to interpret and analyse the models, need to be capable of processing and evaluating the adapted software, hardware, and performance models.*

In CB_3 (see Chapter 8), we adopted the solver SimuLizar, in a way that the solver can interpret and analyse the memory architecture model. For CB_1 and CB_2 no adaptation of the solver was required, since we did not alter the PCM here.

Open Tasks: Currently there remains no open task here. If we tackle the previously stated open task, we might need to reconsider altering the solvers.

10.2.2.4. Assess $R_{accuracy}$

> *$R_{accuracy}$: The performance predictions need to align with the real and measurable behaviour of the software to an extent that is useful for the software architect.*

In CB_2, CB_3, and CB_4 we faced the requirement and aimed for an improvement of performance predictions. With both CB_2 and CB_3 we can provide an approach that greatly increases the accuracy of performance predictions for parallel applications—up to 98% in the best case when using performance curves, and up to 93% accuracy in the best case when using memory modelling.

Open Task: Even though we can increase the predictions, there is still room for improvement. On the one hand, we need to include further PPiFs into the memory models and consider pre-fetching, inter-core communication, and latencies. On the other hand, we need more fine-grain performance curves.

10.2.3. Assess the Research Goal Fulfilment

Given the answers to the research questions and the requirement assessment, we can state that this work has contributed to the improvement of performance predictions for parallel applications in multicore environments. Thereby we have provided better software (i.e., including memory accesses), hardware (i.e., including memory hierarchies), and performance prediction models (i.e., adopting SimuLizar, using CPU Simulators, and providing performance curves). Further, we have contributed to the usability aspect by providing a parallel architectural template catalogue.

Even though we have identified several open questions for future work, we did achieve our research goal, contribute to the domain of SPE, and enable (and improve) performance predictions for parallel applications in multicore environments.

11. Conclusion & Future Work

In the final chapter, we recap the most important insights given in the contributions CB_1 to CB_4. Thereby, we briefly summarise the method, findings, and outcome. Further, in this chapter we discuss the open challenges and remaining tasks for future work in detail. We do not discuss threats to validity separately. However, we did discuss the threats to validity for each contribution in detail in the corresponding chapters, and refer to the sections 6.8 (CB_1), 7.8 (CB_2), 8.6 (CB_3), and 9.6.4 (CB_4).

11.1. Conclusion

Software-rich applications dominate our daily life more and more. These applications fulfil complex and tasks critical to safety. Therefore, it is essential that the application comply with high-quality standards and meet SLO. To ensure high-quality standards, we have to develop software in an engineering-like manner.

One aspect of software engineering is model-based performance prediction, in which software architects model software architectures, enrich the models with performance-relevant information, and use analytical or simulation-based solvers to predict quality attributes, such as performance on architectural drafts during the early design phase. Current state-of-the-art model-based performance prediction approaches can give accurate predictions for even complex systems. To do so, they consider the user's behaviour, software behaviour, and hardware characteristics. For the latter, they only consider CPU-speed as a single metric.

However, modern processor architectures consist of multiple CPU cores, complex memory architectures, and extensive optimisation mechanisms. To fully utilise such multicore architectures, software developers have to

develop the software in a parallel manner, which is even more complicated and makes an engineering-like approach more relevant than ever. However, since model-based performance prediction approaches only consider CPU speed—which by now is no longer the only limiting factor—the accuracy of predictions for parallel applications in multicore environments suffers greatly.

To support SA in making accurate performance predictions for parallel applications, we researched applications for parallel performance predictions in this thesis. Thereby we faced the requirements $R_{modelling}$, R_{metric}, $R_{performance}$, $R_{solvers}$, and $R_{accuracy}$ (see Chapter 1).

As a contribution regarding the requirement $R_{modelling}$, we present a parallel performance pattern catalogue to the SA (see Chapter 6). The pattern catalogue enables the SA to (a) specify the behaviour of highly parallel applications in software models, and (b) to reduce the time and effort needed.

As a contribution regarding the requirement $R_{metrics}$, we present a memory meta-model which includes the most relevant memory hierarchy characteristics (see Chapter 8). Further, we included the meta-model as a meta-model extension in the PCM, and provided graphical editors to the SA to model memory hierarchies in the hardware model, and memory behaviour in the software models.

As a contribution regarding the requirement $R_{performance}$, we present a set of performance curves to the SA (see Chapter 7). The performance curves reflect the speedup behaviour of the six most common resource demand types. Thus, with the help of the performance curves, the SA can consider the speedup behaviour of a parallel application in the prediction models. Thereby, the performance curves can be quickly added and provide a high-level view of complex correlation.

As a contribution regarding the requirement $R_{solvers}$, we extended a performance prediction solver SimuLizar to interpret and analyse the memory meta-model (see Chapter 8). Thus, we enabled the SA to analyse complex memory hierarchies typical in multicore CPUs. Further, we give a proof-of-concept approach on how to include CPU simulators in the workflow of performance predictions (see Chapter 9)

Finally, as a contribution regarding the requirement $R_{accuracy}$, we evaluated the performance curves, memory hierarchy modelling, and CPU simulators

against various use cases. As a result, we are able to show that both the memory hierarchy modelling and the performance curves contribute to the predictive power. Thereby, both approaches contribute and work best in specific scenarios. We can achieve an accuracy of up to 98% in the best case when using performance curves, and up to 93% accuracy in the best case when using memory modelling.

So, to wrap it up, we provide new tools to the SA's silver box. These tools enable him to model the behaviour of highly parallel systems in software performance models, let him specify the characteristics of multicore environments in the hardware performance model, and give him enhanced model-based performance solvers to achieve more accurate performance predictions for parallel applications in multicore environments.

These tools help SAs to create and evaluate high-quality software architectures, which meet the SLOs, already during the design phase.

11.2. Future Work

In the course of this thesis, we researched multiple approaches to enable the software architect to better handle performance prediction for parallel applications. Even though we answered all our research questions and made a significant step in the direction of requirements fulfilment, we also raised new questions, research ideas, and an approach to be even better in the sense of requirement fulfilment. In the following, we briefly sketch the open challenges left for future work. Thereby, we group the items according to the contributions.

CB_1: Parallel Architectural Template Catalogue In CB_1, we researched the modelling language capabilities regarding their suitability for similar behaviour. As a result, we introduced a parallel pattern catalogue based on the AT method. In the first step, we only focused on thread-based patterns.

Thus, a challenge for future work is to investigate other patterns that also represent parallelisation paradigms, such as message passing (e.g., MPI or Actors).

Futher, the current approach supports an abstract method to include the overhead of parallel applications (e.g., forking or synchronisation). With performance curves, we give the SA a tool to make the overhead estimation simple. However, different tool and language support is desirable to reduce the abstraction level, and to make it more precise.

Additionally, the current approaches neglect inter-thread communication, even though inter-thread communication is a relevant PPiF as well. Thus, an additional challenge is to include concepts to simplify the complex patterns of inter-thread communication, and to fit them into the modelling languages.

When it comes to evaluation, the empirical study already gives strong evidence. However, further studies with larger sample sizes and more complex use cases could help to collect additional insights.

CB_2: Parallel Performance Curves In the evaluation of CB_2, we saw that performance curves already improve the predictive power of performance prediction approaches. However, depending on the scenario, the prediction error is still higher than our overall goal of 20%. Thus, we need to reconsider the choice of PPiFs and the use of synthetic demands in future work. Further, a more fine-grained categorisation or other performance curves could contribute to a better result.

Further, a model which allows the SA to specify the sequential and parallel part of an application (e.g., following Amdahl's law) and the specification of the I/O and processor-intensive share (e.g., a demand type which contains 20% of I/O-intensive and 80% of processor-intensive demands) would be beneficial for a better characterisation of resource demands.

Another aspect is evaluation. We evaluate the approach using the SPEC benchmark, which covers a comprehensive set of representative demands. However, using a real-world example, e.g., simulations for material science, might offer further insights.

CB_3: Memory Model Extension for the PCM In CB_3, we extended the PCM to include memory hierarchies and memory behaviour. Due to the very complex characteristics of memory behaviour, this is one of the most challenging endeavours. The approach we present in CB_3 is a first step, in which

we simplified some of the complex interaction of hardware, software, and controller.

By simplifying (abstracting) the memory effects, we did not consider data locality, workload balance and NUMA nodes. To give an example for the latter, each NUMA node has its own architecture, which is characterised by a fast bandwidth. However, when accessing the data from another NUMA node, another much slower bandwidth is used. This can greatly affect performance. Next, we included the concept of latency in our models, but did not further investigate latency effects in memory. We also did not explore snooping or cache-coherency effects. Thus, by setting memory bandwidth latencies and considering cache coherency effects, the performance predictions could benefit. Additionally, current CPUs use pre-fetchers to avoid cache misses and to give performance boosts. We considered these effects in the abstract form of cache hit rates, but a more proactive approach might be needed. Finally, we have not yet combined the memory model with the parallel AT catalogue. A combination would give the SA additional comfort and freedom.

When it comes to evaluation, we conducted a level 1—proof of concept evaluation—using one use case. This evaluates the memory model extension, but the scientific power regarding the prediction accuracy is relatively weak. Thus, further comparisons with more complex examples will help to make more fine-grained models, and give a better understanding of the predictive power.

CB_4: CPU Simulators In CB_4, we adopted the Palladio Bench workflow to transform the PCM models in a running performance prototype, which we then fed into multicore CPU simulators to gain more accurate performance predictions. The approach seems very promising. However, the results we achieved were often highly inaccurate.

A significant challenge for future work is adaptability. Only a few CPU simulators support native Java applications as input, and the ones that do require a Java version below 1.8. This results in significant compatibility issues.

Nevertheless, we rate the insights we gained as very relevant. Therefore, transferring the concepts from CPU simulators at least to some extent into performance simulators, such as SimuLizar, sounds very promising.

Further, a factor involved in the low accuracy could be the simplified PCM models. Thus, including additional model elements, as we did in CB_3, might lead to better results when also adopting ProtoCom.

Also, the evaluation was carried out with a single use case, and served as a proof-of-concept evaluation. We might achieve better results and further insights by using additional use cases.

A. Appendix

A.1. Publications & Supervised Theses

In the context of this doctoral project, we published a number of peer-reviewed publications including conference papers, journals, workshops, and posters. Figure A.1 indicates (in blue) the publications for each area of the thesis.

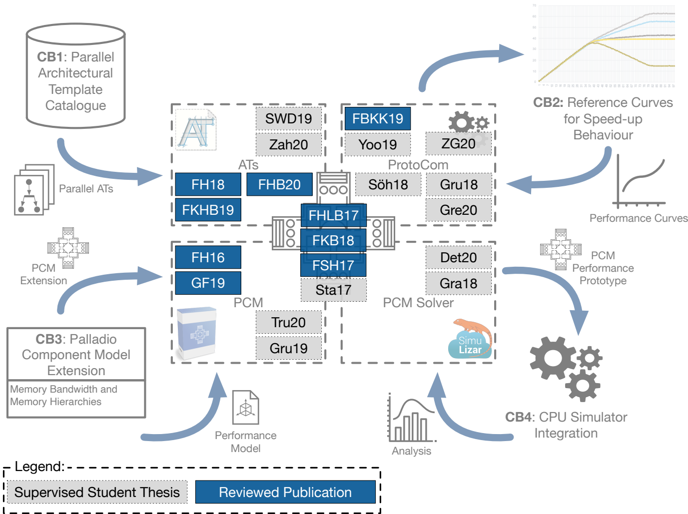


Figure A.1.: Publications:[FBKK19; FH16; FH18; FHLB17; FKB18; FKHB19; FSH17; GF19], Theses: [Det20; Gra18; Gre19; Gru19; Gru20; Söh18; Sta17; SWD19; Tru20; Yoo19; Zah20]

Further, a number of student thesis were supervised by the author of this thesis. We highlight the supervised theses (in grey) and map each one to the areas it addresses.

A.2. Implementations of Resource Demands in Protocom

A.2.1. Fibonacci Numbers

In comparison to the example given in Chapter 5, Protcom uses an iterative approach (see Lst. A.1). This implementation does not focus on a specific Fibonacci number, but on the number of Fibonacci calculations performed (given bei the `iterationCount`).

```
1  private long fibonacci(double iterationCount) {
2        long i1 = 1;
3        long i2 = 1;
4        long i3 = 0;
5        for (long i = 0; i < iterationCount; i++) {
6           i3 = i1 + i2;
7           i2 = i1;
8           i1 = i3;
9        }
10       return i3;
```

Listing A.1: Implementation of the Fibonacci demand in Protocom

A.2.2. Mandel Set

```
1     private void drawMandelbrot(long init) {
2        // Date d1 = new Date();
3        int n = (int) init;
4        float m = n;
5        int x, y;
6        for (y = -n; y < n; y++) {
7           // System.out.print("\n");
8           for (x = -n; x < n; x++) {
9              if (iterate(x / m, y / m) == 0) {
10                // System.out.print("*");
11             } else {
12                // System.out.print(" ");
13             }
```

```
14
15              }
16          }
17          // Date d2 = new Date();
18          // long diff = d2.getTime() - d1.getTime();
19          // System.out.println("\nJava Elapsed " + diff / 1000.0f);
20      }
21
22      private int iterate(float x, float y) {
23          float cr = y - 0.5f;
24          float ci = x;
25          float zi = 0.0f;
26          float zr = 0.0f;
27          int i = 0;
28          while (true) {
29              i++;
30              float temp = zr * zi;
31              float zr2 = zr * zr;
32              float zi2 = zi * zi;
33              zr = zr2 - zi2 + cr;
34              zi = temp + temp + ci;
35              if (zi2 + zr2 > BAILOUT) {
36                  return i;
37              }
38              if (i > MAX_ITERATIONS) {
39                  return 0;
40              }
41          }
42      }
```

Listing A.2: Implementation of the Mandel Set demand in Protocom

A.2.3. Sorting Arrays

```
1   public SortArrayDemand(final int arraySize) {
2       super(-3, 0, 3, 10000, 50);
3       this.arraySize = arraySize;
4       this.values = new double[this.arraySize];
5       final Random r = new Random(SEED);
6       for (int i = 0; i < this.values.length; i++) {
7           this.values[i] = r.nextDouble();
8       }
9   }
10
11  public SortArrayDemand() {
12      this(DEFAULT_ARRAY_SIZE);
13  }
14
15  private void sortArray(final int amountOfNumbers) {
16      final int iterations = amountOfNumbers / this.arraySize;
17      final int rest = amountOfNumbers % this.arraySize;
18      for (int i = 0; i < iterations; i++) {
```

```
19            final double[] lotsOfDoubles = getArray(this.arraySize);
20            Arrays.sort(lotsOfDoubles);
21         }
22         final double[] lotsOfDoubles = getArray(rest);
23         Arrays.sort(lotsOfDoubles);
24      }
```

Listing A.3: Implementation of the Sorting Array demand in Protocom

A.2.4. Calculate Prime Demand

```
1       private long calculatePrime(double numberNextPrimes) {
2
3          boolean isPrime = true;
4          long currentNumber = number;
5          long primesFound = 0;
6          long currentDivisor;
7          long upperBound;
8
9          while (primesFound < numberNextPrimes) {
10            // test primality of currentNumber
11            currentDivisor = 2;
12            upperBound = currentNumber / 2;
13            while ((currentDivisor < upperBound) && (isPrime)) {
14               isPrime = currentNumber % currentDivisor != 0;
15               currentDivisor++;
16            }
17            // count primes and continue
18            if (isPrime) {
19               primesFound++;
20            }
21            // prepare for next iteration
22            isPrime = true;
23            currentNumber++;
24         }
25         return currentNumber;
26      }
```

Listing A.4: Implementation of the Sorting Array demand in Protocom

A.2.5. Counting Numbers Demand

```
1    private void countNumbers(double countTo) {
2          for (long j = 0; j < countTo; j++) {
3             if (k > 100000) {
4                k = 0;
5             }
6             k += j;
```

```
7        }
```

Listing A.5: Implementation of the Counting Numbers demand in Protocom

A.2.6. Matrix Multiplicationn Demand

```
1   private static final int DEFAUL_MATRIX_SIZE = 500;
2   private final double matrixA[][];
3   private final double matrixB[][];
4   private final int matrixSize;
5
6   public MultiplyMatrixDemand(int matrixSize) {
7     super(-3, 0, 3, 10000, 50);
8     this.matrixSize = matrixSize;
9
10    matrixA = new double[matrixSize][matrixSize];
11    matrixB = new double[matrixSize][matrixSize];
12
13    fillMatrixRandom(matrixA);
14    fillMatrixRandom(matrixB);
15
16  }
17
18  public MultiplyMatrixDemand() {
19    this(DEFAUL_MATRIX_SIZE);
20  }
21
22  private void multiplyMatrix(final long numberOfMultiplications) {
23    double resultMatrix[][] = new double[matrixSize][matrixSize];
24    long numberOfPerformedMultiplications = 0;
25
26    while (numberOfPerformedMultiplications < numberOfMultiplications) {
27      for (int i = 0; i < matrixA.length; i++) {
28        for (int k = 0; k < matrixB.length; k++) {
29          for (int j = 0; j < matrixA.length; j++) {
30            if(numberOfPerformedMultiplications < numberOfMultiplications) {
31              resultMatrix[i][j] = resultMatrix[i][j] + matrixA[i][k] * matrixB[k][j];
32              numberOfPerformedMultiplications++;
33            }else {
34              return;
35            }
36          }
37        }
38      }
39    }
40  }
```

Listing A.6: Implementation of the Matrix Multiplication demand in Protocom

A.3. User Study Protocols

A.3.1. Blank User Study Leaflet—Group A

Controlled User Study: Usability and Efficiency Evaluation of the Parallel Performance Catalogue Extension for the Palladio-Bench

User Study Leaflet

General Information:

In this experiment you will be modeling parallel behaviors in Palladio. The experiment contains two use case scenarios and each scenario contains one modeling task. For each task you will have 30 minutes. In order for your participation to be successful you have to work on both tasks. You modeling solution is correct when a simulation of the model starts and finishes successfully. Even if you are not able to achieve a working model in the given time, your submission still counts and your participation will be counted as successful. While you are completing the modeling tasks, your task completion time, number of errors, and time spent in errors will be recorded and noted. At certain points during the study, you will encounter questions from the questionnaire which you have to answer before proceeding with the next task.

Introductory questions:

1. Your current academic degree is: ______________________________
2. How would you rate your experience in the field of performance engineering?

   none ☐ ☐ ☐ ☐ ☐ ☐ ☐ expert

3. How would you rate your experience with Palladio before the conduction of this experiment?

   none ☐ ☐ ☐ ☐ ☐ ☐ ☐ expert

Consent Form

DESCRIPTION: You are invited to participate in **a research study** on **different modeling tools in the Palladio-Bench tool**.

TIME INVOLVEMENT: Your participation will take approximately **60 minutes.**

DATA COLLECTION: For this study you will model use case scenarios in Palladio. During the modeling process, metrics such as task completion time, number of errors and time spent in errors will be measured. Also, you will need to fill in a questionnaire.

RISKS AND BENEFITS: No risk associated with this study. The collected data is securely stored. We do guarantee no data misuse and privacy is completely preserved. Your decision whether or not to participate in this study will not affect your grade in school.

PARTICIPANT'S RIGHTS: If you have read this form and have decided to participate in this project, please understand your **participation is voluntary** and you have the **right to withdraw your consent or discontinue participation at any time without penalty or loss of benefits to which you are otherwise entitled**. **The alternative is not to participate.** The results of this research study may be presented at scientific or professional meetings or published in scientific journals. Your identity is not disclosed unless we directly inform and ask for your permission.

CONTACT INFORMATION: If you have any questions, concerns or complaints about this research, its procedures, risks and benefits, contact following persons:
Denis Zahariev (denis.zahariev95@gmail.com)
Markus Frank (markus.frank@iste.uni-stuttgart.de)

By signing this document I confirm that I agree to the terms and conditions.

Name: ______________________ *Signature, Date:* ______________________

Use Case Scenarios and Modeling Tasks

Use Case Scenario 1

Start with reading the use case description and then proceed with the task.

Use Case Description:

The software in this use case is used to search for a list of literature in various scientific databases. The search is executed in parallel where each database is searched in a separate thread. For the purpose of this scenario, the number of databases is limited to 16. The software consists of one component and one providing interface. The interface declares the search method and the component implements it. In the specification of the method create all of the threads responsible for the search. The searching operation for one list of literature in a single database requires 100 CPU resources. Each thread also requires 5 CPU resources for the synchronization overhead resulting from the creation and the start of the thread. Exactly one instance of the component and the interface are present in the software system. The resource environment where the system is deployed has a CPU with a processing rate of 200 and 4 number of replicas and the whole system is deployed on a single container. In the usage scenario, a single call of the search method is started with a closed workload of one user and no think time.

Task A (Standard toolkit):

In the project that you receive every diagram is complete except the SEFF Diagram of the basic component. Your task is to complete the SEFF Diagram.

Questionnaire

Questions regarding Use Case Scenario 1:

4. How would you rate the difficulty of the task in Use Case Scenario 1?

   very easy ☐ ☐ ☐ ☐ ☐ ☐ ☐ very hard

5. How would you rate your performance regarding the task in Use Case Scenario 1?

   very slow ☐ ☐ ☐ ☐ ☐ ☐ ☐ very fast

6. How would you rate the amount of work required for completing the task in Use Case Scenario 1?

   too little ☐ ☐ ☐ ☐ ☐ ☐ ☐ too much

7. How would you rate the usability of the standard toolkit regarding the modeling of parallel behaviors and your user experience with it?

   very bad ☐ ☐ ☐ ☐ ☐ ☐ ☐ very good

Use Case Scenarios and Modeling Tasks

Use Case Scenario 2

Start with reading the use case description and then proceed with the task.

Use Case Description:

The software in this use case is used in machine learning in order to speed up complex calculations. It multiplies two 16x16 matrices and the multiplication is executed in parallel where each row of the resulting matrix is calculated in a separate thread. With the given size of the matrices, this results in 16 threads. The software consists of one component and one providing interface. The interface declares the multiply method and the component implements it. The multiplication operation for one of the resulting rows requires 125 CPU resources. Each thread also requires 5 CPU resources for the synchronization overhead resulting from the creation and the start of the thread. Exactly one instance of the component and the interface are present in the software system. The resource environment where the system is deployed has a CPU with a processing rate of 250 and 4 number of replicas and the whole system is deployed on a single container. In the usage scenario, a single call of the multiply method is started with a closed workload of one user and no think time.

Task B (Parallel Performance Catalogue):

In the project that you receive every diagram is complete except the SEFF Diagram of the basic component. The files required for the experiment automation are also complete. Your task is to complete the SEFF Diagram and to apply the Parallel Loops AT.

Questionnaire

Questions regarding Use Case Scenario 2:

1. How would you rate the difficulty of the task in Use Case Scenario 2?

   very easy ☐ ☐ ☐ ☐ ☐ ☐ ☐ very hard

2. How would you rate your performance regarding the task in Use Case Scenario 2?

   very slow ☐ ☐ ☐ ☐ ☐ ☐ ☐ very fast

3. How would you rate the amount of work required for completing the task in Use Case Scenario 2?

   too little ☐ ☐ ☐ ☐ ☐ ☐ ☐ too much

4. How would you rate the usability of the Parallel Performance Catalogue regarding the modeling of parallel behaviors and your user experience with it?

   very bad ☐ ☐ ☐ ☐ ☐ ☐ ☐ very good

Questions regarding the Parallel Performance Catalogue:

5. How would you rate the usability of the Parallel Performance Catalogue in comparison to the standard toolkit?

   worse than the standard toolkit ☐ ☐ ☐ ☐ ☐ ☐ ☐ significantly better and easier than the standard toolkit

Questionnaire

6. How would you rate the following statement:

   "The Parallel Performance Catalogue introduces a very significant speed-up regarding the modeling of parallel behaviors."

   false ☐ ☐ ☐ ☐ ☐ ☐ ☐ true

7. Would you recommend the usage of the Parallel Performance Catalogue to other users of Palladio?

   definitely no ☐ ☐ ☐ ☐ ☐ ☐ ☐ definitely yes

Final thoughts

8. What did you like about the user experiment?

   ____

   ____

   ____

9. What did you not like about the user experiment?

   ____

   ____

   ____

10. What would you improve about the Parallel Performance Catalogue?

   ____

   ____

   ____

A.3.2. Blank User Study Leaflet—Group B

Controlled User Study: Usability and Efficiency Evaluation of the Parallel Performance Catalogue Extension for the Palladio-Bench

User Study Leaflet

General Information:

In this experiment you will be modeling parallel behaviors in Palladio. The experiment contains two use case scenarios and each scenario contains one modeling task. For each task you will have 30 minutes. In order for your participation to be successful you have to work on both tasks. You modeling solution is correct when a simulation of the model starts and finishes successfully. Even if you are not able to achieve a working model in the given time, your submission still counts and your participation will be counted as successful. While you are completing the modeling tasks, your task completion time, number of errors, and time spent in errors will be recorded and noted. At certain points during the study, you will encounter questions from the questionnaire which you have to answer before proceeding with the next task.

Introductory questions:

1. Your current academic degree is: ______________________________
2. How would you rate your experience in the field of performance engineering?

   none ☐ ☐ ☐ ☐ ☐ ☐ ☐ expert

3. How would you rate your experience with Palladio before the conduction of this experiment?

   none ☐ ☐ ☐ ☐ ☐ ☐ ☐ expert

Consent Form

DESCRIPTION: You are invited to participate in **a research study** on **different modeling tools in the Palladio-Bench tool**.

TIME INVOLVEMENT: Your participation will take approximately **60 minutes.**

DATA COLLECTION: For this study you will model use case scenarios in Palladio. During the modeling process, metrics such as task completion time, number of errors and time spent in errors will be measured. Also, you will need to fill in a questionnaire.

RISKS AND BENEFITS: No risk associated with this study. The collected data is securely stored. We do guarantee no data misuse and privacy is completely preserved. Your decision whether or not to participate in this study will not affect your grade in school.

PARTICIPANT'S RIGHTS: If you have read this form and have decided to participate in this project, please understand your **participation is voluntary** and you have the **right to withdraw your consent or discontinue participation at any time without penalty or loss of benefits to which you are otherwise entitled**. **The alternative is not to participate.** The results of this research study may be presented at scientific or professional meetings or published in scientific journals. Your identity is not disclosed unless we directly inform and ask for your permission.

CONTACT INFORMATION: If you have any questions, concerns or complaints about this research, its procedures, risks and benefits, contact following persons:
Denis Zahariev (denis.zahariev95@gmail.com)
Markus Frank (markus.frank@iste.uni-stuttgart.de)

By signing this document I confirm that I agree to the terms and conditions.

Name: ______________________ *Signature, Date:* ______________________

Use Case Scenarios and Modeling Tasks

Use Case Scenario 1

Start with reading the use case description and then proceed with the task.

Use Case Description:
The software in this use case is used to search for a list of literature in various scientific databases. The search is executed in parallel where each database is searched in a separate thread. For the purpose of this scenario, the number of databases is limited to 16. The software consists of one component and one providing interface. The interface declares the search method and the component implements it. In the specification of the method create all of the threads responsible for the search. The searching operation for one list of literature in a single database requires 100 CPU resources. Each thread also requires 5 CPU resources for the synchronization overhead resulting from the creation and the start of the thread. Exactly one instance of the component and the interface are present in the software system. The resource environment where the system is deployed has a CPU with a processing rate of 200 and 4 number of replicas and the whole system is deployed on a single container. In the usage scenario, a single call of the search method is started with a closed workload of one user and no think time.

Task B (Parallel Performance Catalogue):
In the project that you receive every diagram is complete except the SEFF Diagram of the basic component. The files required for the experiment automation are also complete. Your task is to complete the SEFF Diagram and to apply the Parallel Loops AT.

Questionnaire

Questions regarding Use Case Scenario 1:

1. How would you rate the difficulty of the task in Use Case Scenario 1?

   very easy ☐ ☐ ☐ ☐ ☐ ☐ ☐ very hard

2. How would you rate your performance regarding the task in Use Case Scenario 1?

   very slow ☐ ☐ ☐ ☐ ☐ ☐ ☐ very fast

3. How would you rate the amount of work required for completing the task in Use Case Scenario 1?

   too little ☐ ☐ ☐ ☐ ☐ ☐ ☐ too much

4. How would you rate the usability of the Parallel Performance Catalogue regarding the modeling of parallel behaviors and your user experience with it?

   very bad ☐ ☐ ☐ ☐ ☐ ☐ ☐ very good

Use Case Scenarios and Modeling Tasks

Use Case Scenario 2

Start with reading the use case description and then proceed with the task.

Use Case Description:

The software in this use case is used in machine learning in order to speed up complex calculations. It multiplies two 16x16 matrices and the multiplication is executed in parallel where each row of the resulting matrix is calculated in a separate thread. With the given size of the matrices, this results in 16 threads. The software consists of one component and one providing interface. The interface declares the multiply method and the component implements it. The multiplication operation for one of the resulting rows requires 125 CPU resources. Each thread also requires 5 CPU resources for the synchronization overhead resulting from the creation and the start of the thread. Exactly one instance of the component and the interface are present in the software system. The resource environment where the system is deployed has a CPU with a processing rate of 250 and 4 number of replicas and the whole system is deployed on a single container. In the usage scenario, a single call of the multiply method is started with a closed workload of one user and no think time.

Task A (Standard toolkit):

In the project that you receive every diagram is complete except the SEFF Diagram of the basic component. Your task is to complete the SEFF Diagram.

Questionnaire

Questions regarding Use Case Scenario 2:

5. How would you rate the difficulty of the task in Use Case Scenario 2?

   very easy ☐ ☐ ☐ ☐ ☐ ☐ ☐ very hard

6. How would you rate your performance regarding the task in Use Case Scenario 2?

   very slow ☐ ☐ ☐ ☐ ☐ ☐ ☐ very fast

7. How would you rate the amount of work required for completing the task in Use Case Scenario 2?

   too little ☐ ☐ ☐ ☐ ☐ ☐ ☐ too much

8. How would you rate the usability of the standard toolkit regarding the modeling of parallel behaviors and your user experience with it?

   very bad ☐ ☐ ☐ ☐ ☐ ☐ ☐ very good

Questions regarding the Parallel Performance Catalogue:

9. How would you rate the usability of the Parallel Performance Catalogue in comparison to the standard toolkit?

   worse than the standard toolkit ☐ ☐ ☐ ☐ ☐ ☐ ☐ significantly better and easier than the standard toolkit

Questionnaire

10. How would you rate the following statement:

"The Parallel Performance Catalogue introduces a very significant speed-up regarding the modeling of parallel behaviors."

false ☐ ☐ ☐ ☐ ☐ ☐ ☐ true

11. Would you recommend the usage of the Parallel Performance Catalogue to other users of Palladio?

definitely no ☐ ☐ ☐ ☐ ☐ ☐ ☐ definitely yes

Final thoughts

12. What did you like about the user experiment?

____

____

____

13. What did you not like about the user experiment?

____

____

____

14. What would you improve about the Parallel Performance Catalogue?

____

____

____

A.3.3. Blank Measurement Protocol

Measurement Protocol

Date: ________________________

Use Case Scenario 1:

1. Start time: ____________________
2. Finish time: ___________________
3. Number of errors and time spent in errors:
   - Total number of errors: ______________________
   - Total time spent in errors: ____________________

Error number	Occurrence	Removal	Duration
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Use Case Scenario 2:

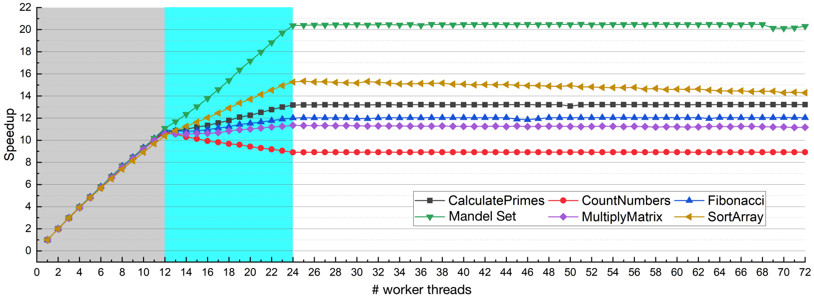
4. Start time: ____________________
5. Finish time: ___________________
6. Number of errors and time spent in errors:
   - Total number of errors: ______________________
   - Total time spent in errors: ____________________

Error number	Occurrence	Removal	Duration
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

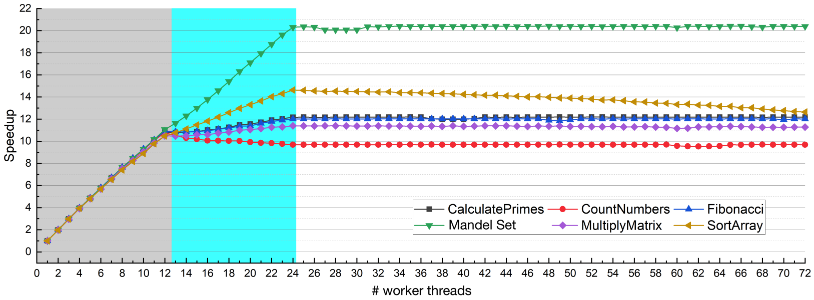
A.4. Additional Performance Factor Measurements

A.4.1. Speedup Behaviour

A.4.1.1. Server Potsdam Small

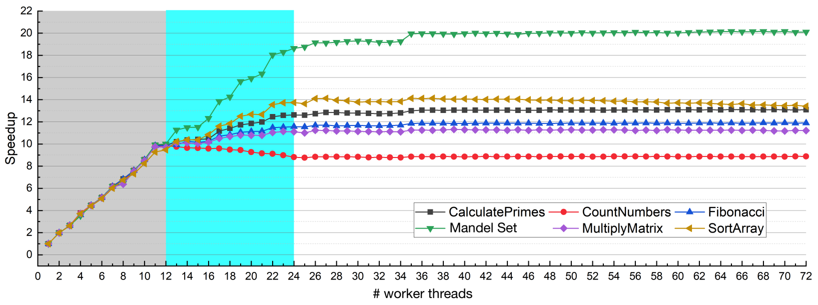


(a) Speedup Curve for all Demands Using Java Threads

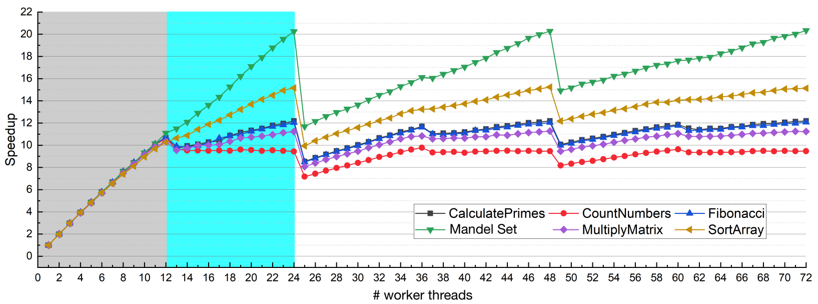


(b) Speedup Curve for all Demands Using Pyjama (OpenMP)

Figure A.2.: Speedup for Threads and OpenMP [Gre19]



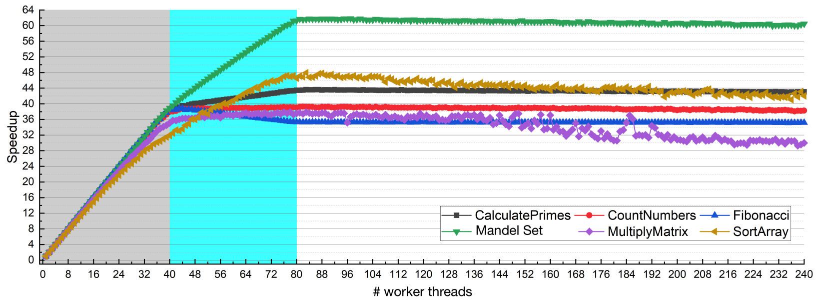
(a) Speedup Curve for all Demands Using Java Streams



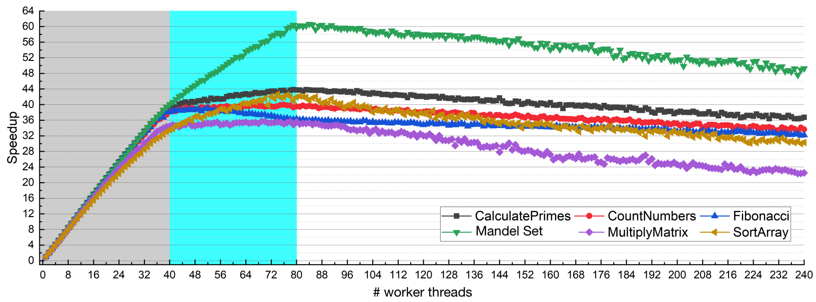
(b) Speedup Curve for all Demands Using AKKA Actors

Figure A.3.: Speedup for Streams and Actors [Gre19]

A.4.1.2. Server Potsdam Large

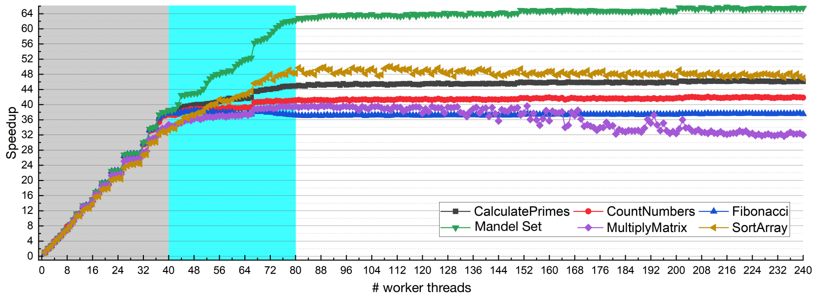


(a) Speedup Curve for all Demands Using Java Threads

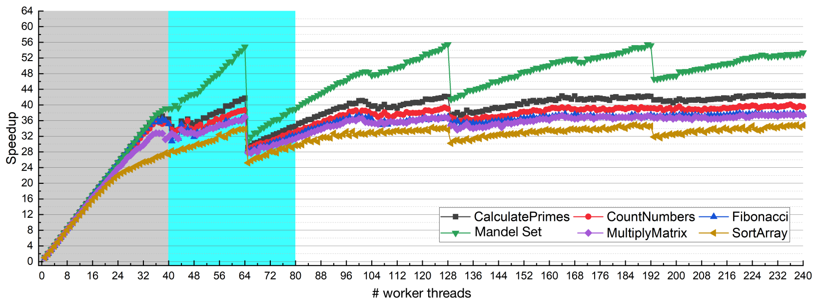


(b) Speedup Curve for all Demands Using Pyjama (OpenMP)

Figure A.4.: Speedup for Threads and OpenMP [Gre19]



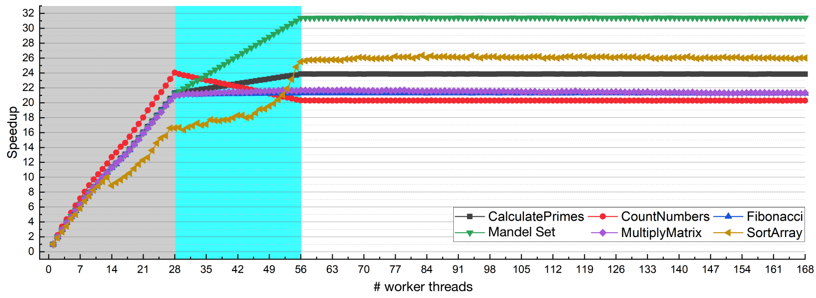
(a) Speedup Curve for all Demands Using Java Streams



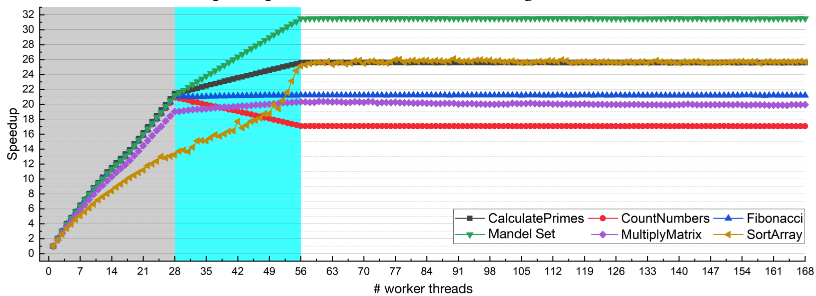
(b) Speedup Curve for all Demands Using AKKA Actors

Figure A.5.: Speedup for Streams and Actors [Gre19]

A.4.1.3. Multi Node Cluster – BW Cloud

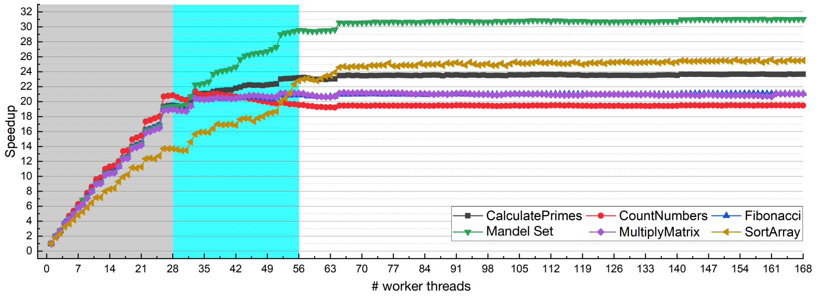


(a) Speedup Curve for all Demands Using Java threads

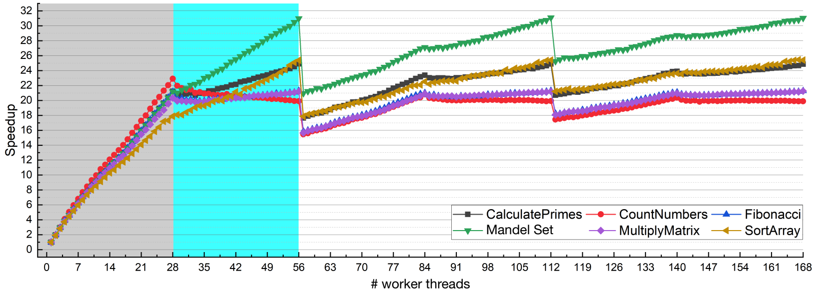


(b) Speedup Curve for all Demands Using Pyjama (OpenMP)

Figure A.6.: Speedup for Threads and OpenMP [Gre19]



(a) Speedup Curve for all Demands Using Java Streams

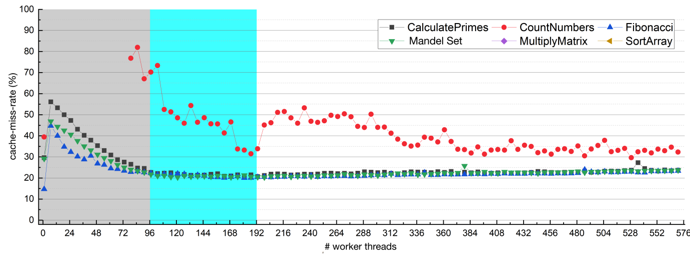


(b) Speedup Curve for all Demands Using AKKA Actors

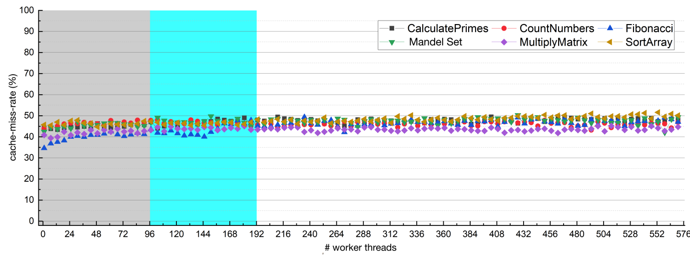
Figure A.7.: Speedup for Streams and AKKA Actors [Gre19]

A.4.2. Cache Behaviour

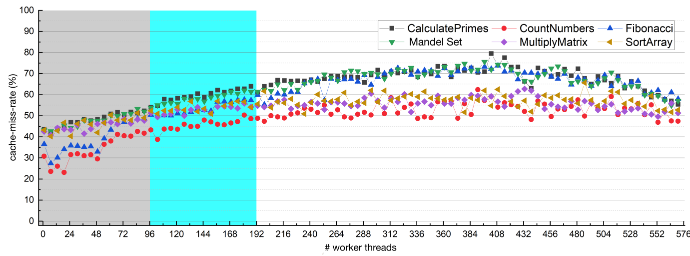
A.4.2.1. Uni Stuttgart – L2 Cache



(a) L2 Cache Behaviour for Pyjama (OpenMP)



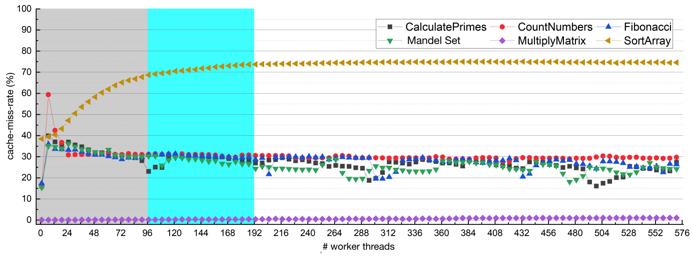
(b) L2 Cache Behaviour for Java Streams



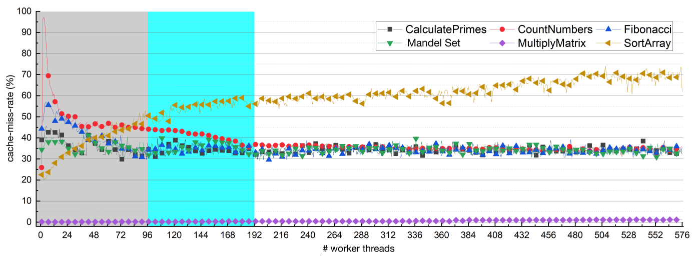
(c) L2 Cache Behaviour for AKKA Actors

Figure A.8.: L2 Cache Behaviour [Gre19]

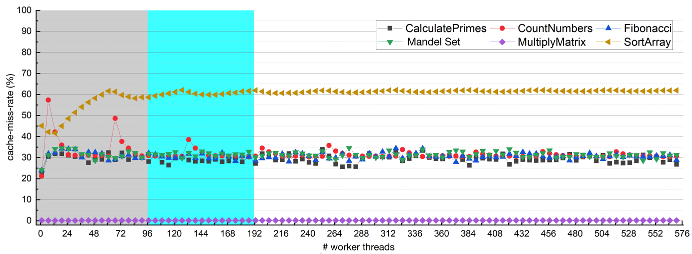
A.4.2.2. Uni Stuttgart – L3 Cache



(a) L3 Cache Behaviour for Pyjama (OpenMP)



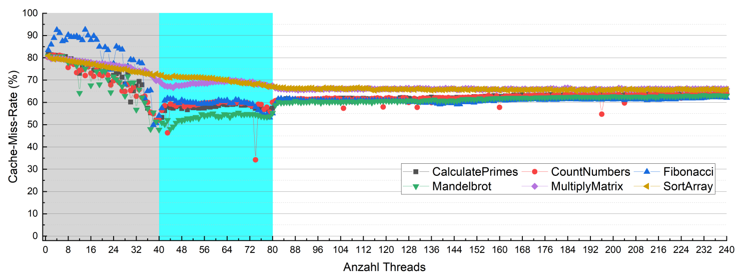
(b) L3 Cache Behaviour for Java Streams



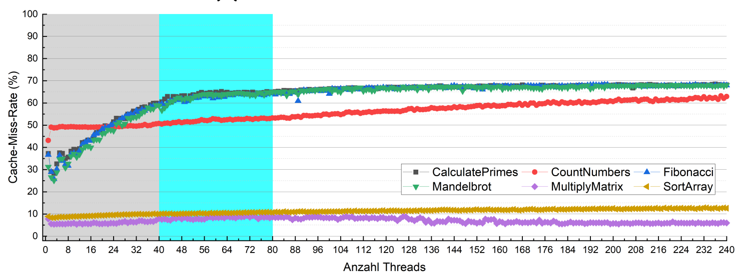
(c) L3 Cache Behaviour for AKKA Actors

Figure A.9.: L3 Cache Behaviour [Gre19]

A.4.2.3. Server Potsdam Large – L2 Cache

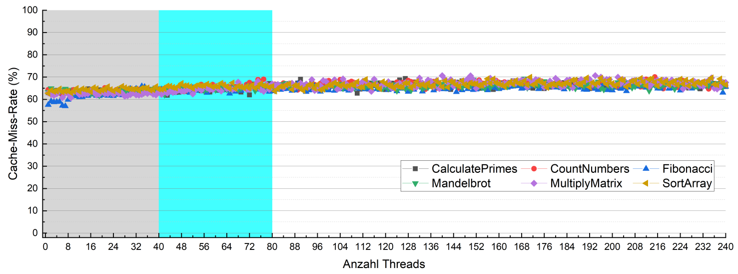


(a) L2 Cache Behaviour for Threads

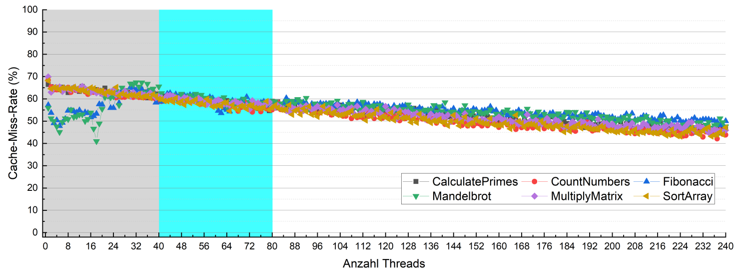


(b) L2 Cache Behaviour for Pyjama (OpenMP)

Figure A.10.: L2 cache behaviour for Threads and OpenMP [Gre19]



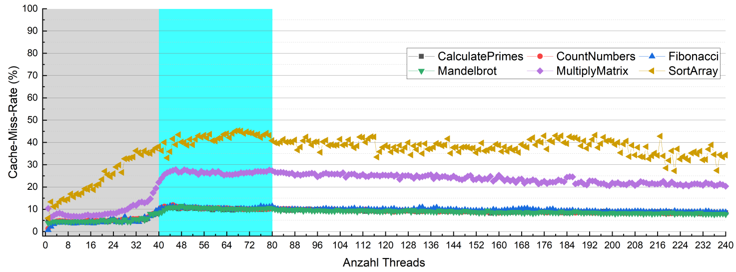
(a) L2 Cache Behaviour for Java Streams



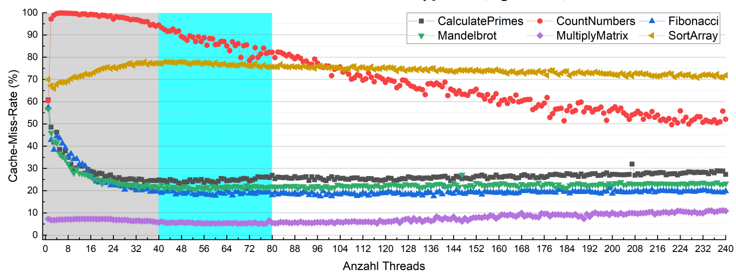
(b) L2 Cache Behaviour for AKKA Actors

Figure A.11.: L2 Cache Behaviour for Streams and Actors [Gre19]

A.4.2.4. Server Potsdam Large – L3 Cache

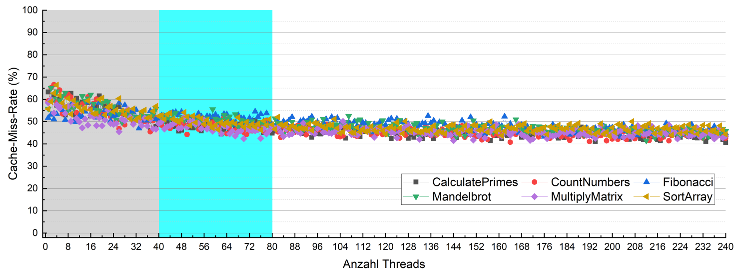


(a) L3 Cache Behaviour for Pyjama (OpenMP)

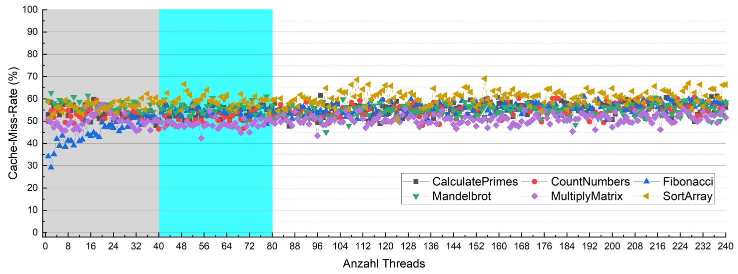


(b) L3 Cache Behaviour for Pyjama (OpenMP)

Figure A.12.: L3 Cache Behaviour for Threads and OpenMP [Gre19]



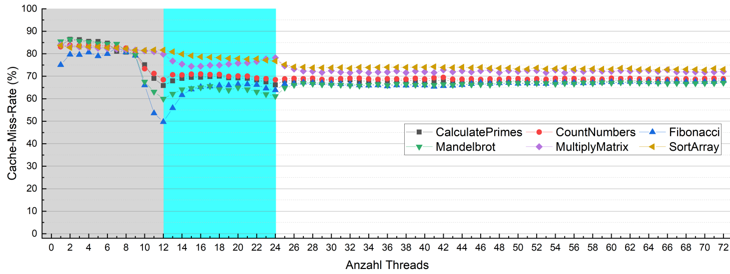
(a) L3 Cache Behaviour for Java Streams



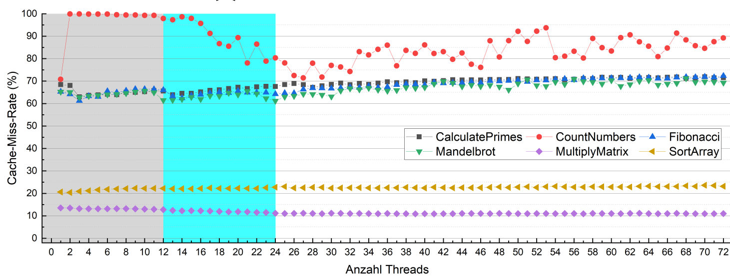
(b) L3 Cache Behaviour for AKKA Actors

Figure A.13.: L3 Cache Behaviour for Streams and AKKA Actors [Gre19]

A.4.2.5. Server Potsdam Small – L2 Cache

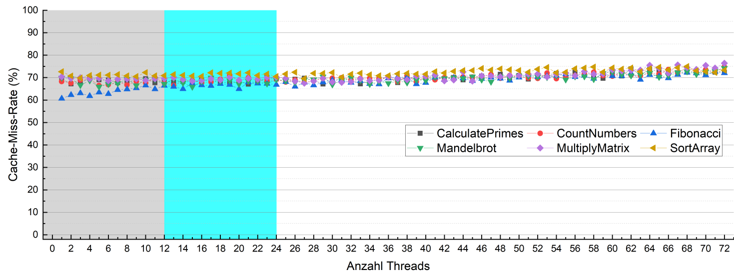


(a) L2 Cache Behaviour for Threads

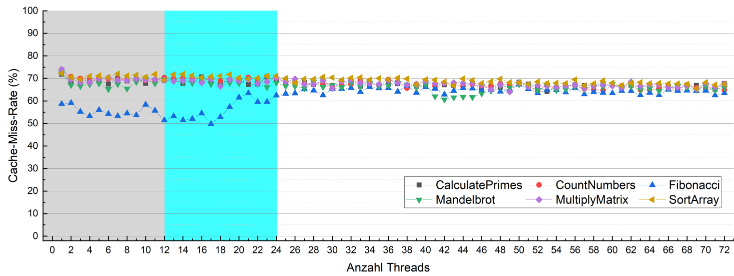


(b) L2 Cache Behaviour for Pyjama (OpenMP)

Figure A.14.: L2 Cache Behaviour for Threads and OpenMP [Gre19]



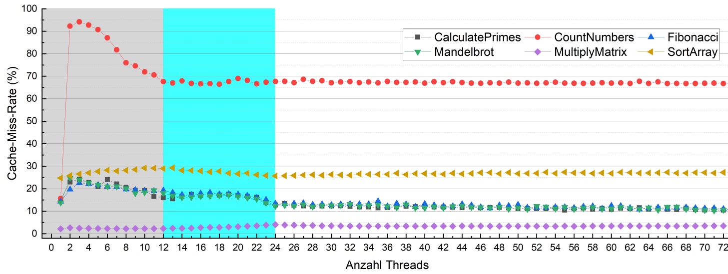
(a) L2 Cache Behaviour for Java Streams



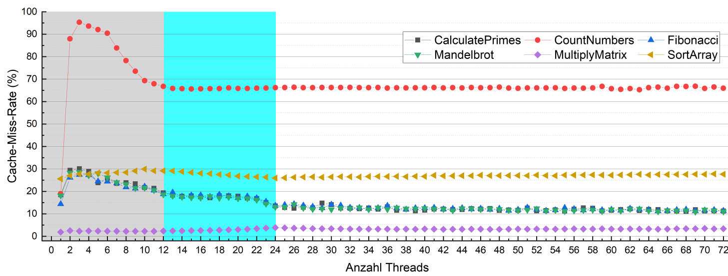
(b) L2 Cache Behaviour for AKKA Actors

Figure A.15.: L2 Cache Behaviour for Streams and AKKA Actors [Gre19]

A.4.2.6. Server Potsdam Small – L3 Cache

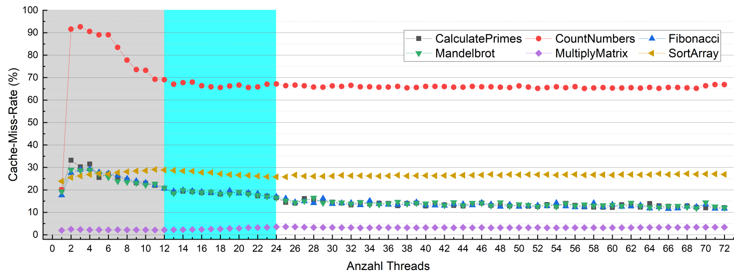


(a) L3 Cache Behaviour for Threads

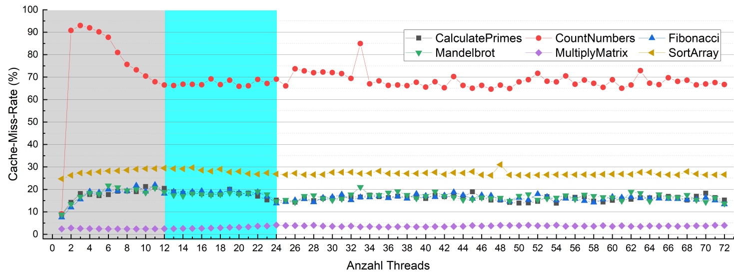


(b) L3 Cache Behaviour for Pyjama (OpenMP)

Figure A.16.: L3 cache behaviour Threads and Streams [Gre19]



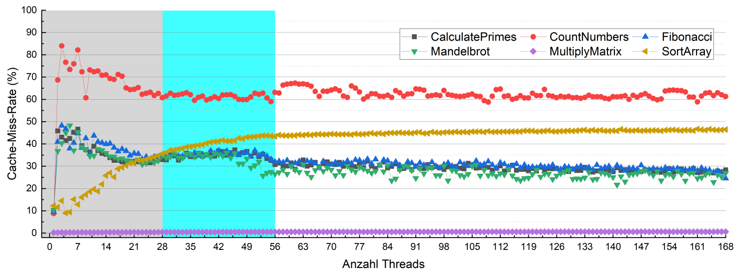
(a) L3 Cache Behaviour for Java Streams



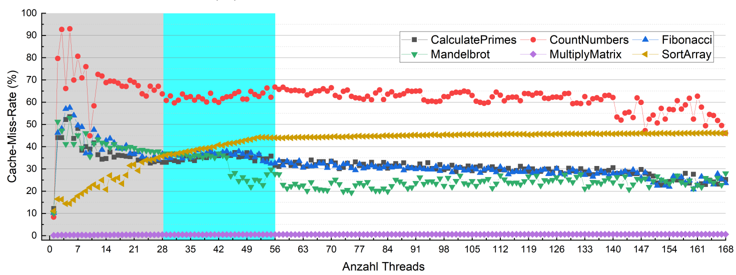
(b) L3 Cache Behaviour for AKKA Actors

Figure A.17.: L3 cache behaviour for Streams and AKKA Actors [Gre19]

A.4.2.7. Multi Node Cluster (BW Cloud) – L3 Cache

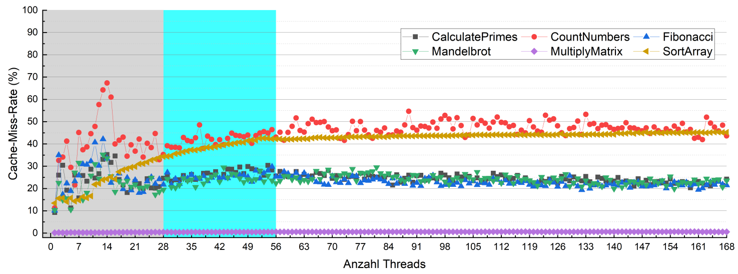


(a) L3 Cache Behaviour for Threads

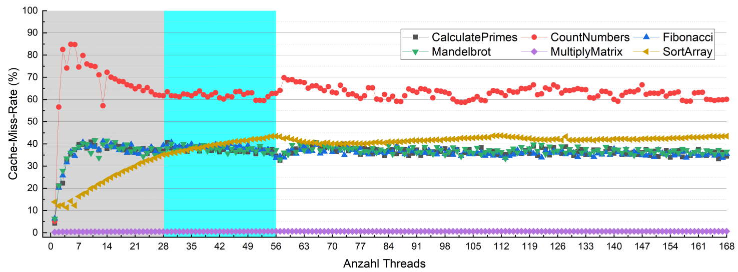


(b) L3 Cache Behaviour for Pyjama (OpenMP)

Figure A.18.: L3 Cache Behaviour for Threads and OpenMP [Gre19]



(a) L3 Cache Behaviour for Java Streams



(b) L3 Cache Behaviour for AKKA Actors

Figure A.19.: L3 Cache Behaviour for Streams and AKKA Actors [Gre19]

A.4.3. Performance Curves

A.4.3.1. Performance Curve for Dedicated Hardware

	$f(x)$ for **Stage**		
Demand Type	**1**	**2**	**3**
CountNumbers	$0.438x$	$-0.171x + 0.572$	$-0.0038x + 0.230$
MatrixMultiplication	$0.412x$	$0.043x + 0.357$	$-0.0148x + 0.472$
FibonacciNumbers	$0.452x$	$0.026x + 0.417$	$0.00341x + 0.456$
PrimeNumbers	$0.449x$	$0.096x + 0.333$	$0.00140x + 0.536$
SortArray	$0.407x$	$0.151x + 0.252$	$-0.0129x + 0.573$
MandelSet	$0.458x$	$0.314x + 0, 206$	$0.00940x + 0.791$

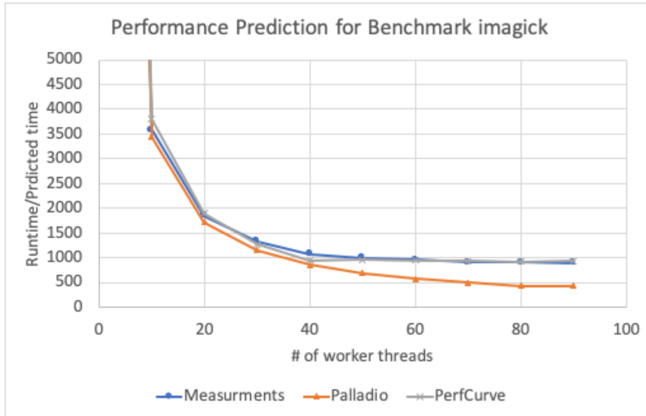
Table A.1.: Extracted Performance Curves for Dedicated Machines based on the Speedup Behaviour of the Demands

A.4.3.2. Performance Curves for Virtualised Hardware

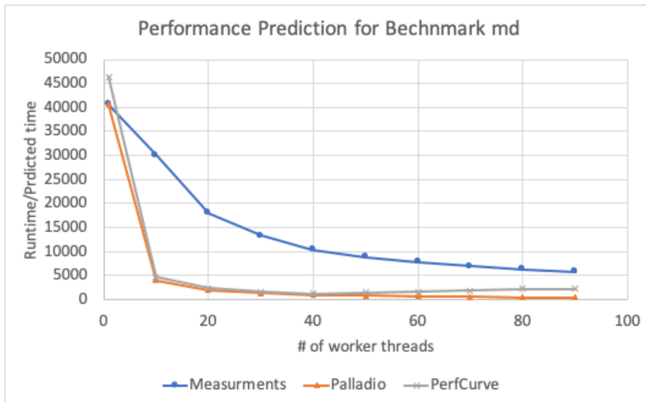
	$f(x)$ for **Stage**		
Demand Type	**1**	**2**	**3**
CountNumbers	$0.374x$	$-0.052x + 0.445$	$-0.0002x + 0.336$
MatrixMultiplication	$0.334x$	$0.0520x + 0.332$	$-0.0006x + 0.373$
FibonacciNumbers	$0.357x$	$0.0096x + 0.359$	$0.03220x + 0.280$
PrimeNumbers	$0.353x$	$0.0610x + 0.308$	$0.00322x + 0.425$
SortArray	$0.241x$	$0.1480x + 0.095$	$0.01800x + 0.404$
MandelSet	$0.349x$	$0.1830x + 0.184$	$0.00870x + 0.532$

Table A.2.: Extracted Performance Curves for Virtualised Machines Based on the Speedup Behaviour of the Demands

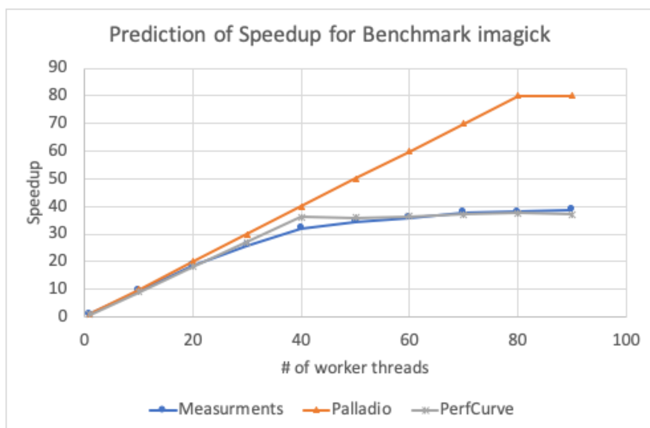
A.4.4. Performance Prediction Error



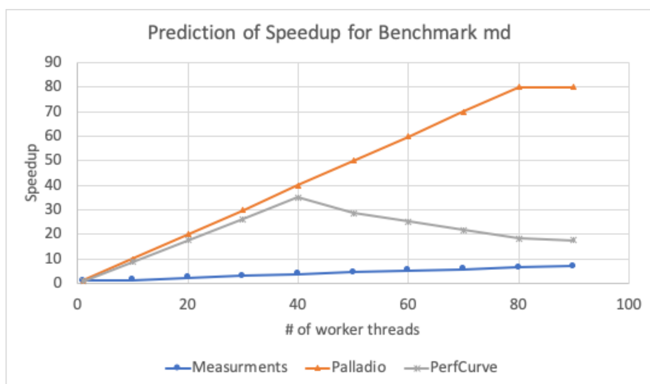
(a) Prediction of the Palladio and Performance Curves in Compression to the Measurements for the Best Case imagick



(b) Prediction of the Palladio and Performance Curves in Compression to the Measurements for the Worst Case md



(a) Prediction of the Speedup for the Approaches Palladio and Performance Curves in Compression to the Measured Speedup for the Best Case imagick



(b) Prediction of the Speedup for the Approaches Palladio and Performance Curves in Compression to the Measured Speedup for the Worst Case md

A.5. Memory Hierarchy Models

A.5.1. Sirius Extension for Memory Hierarchy Model

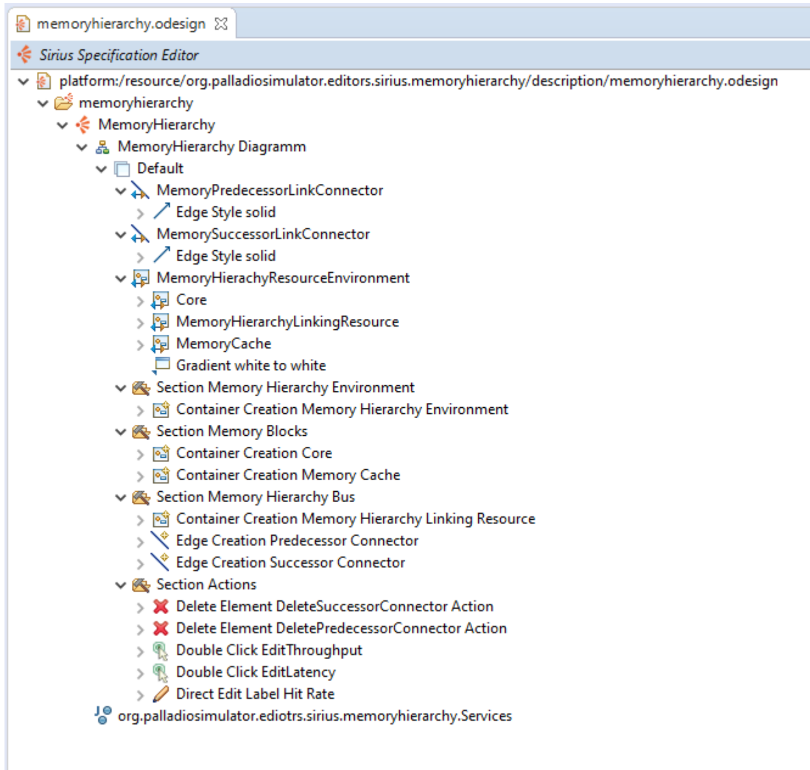


Figure A.22.: Screenshoot of the .odesign File for the Memory Hierarchy [Tru20]

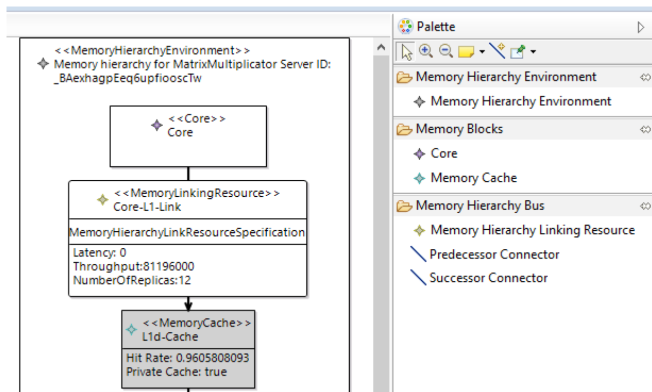


Figure A.23.: Screenshot of the Memory Hierarchy Editor with Palette Showing Elements That Can Be Added to the Diagram [Tru20]

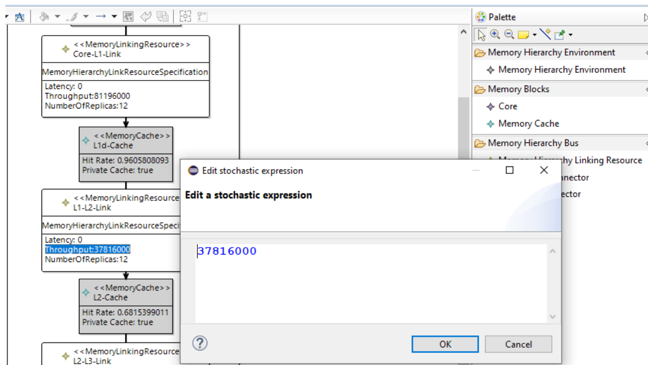


Figure A.24.: Screenshot of the Memory Hierarchy Editor with an Edit Dialog [Tru20]

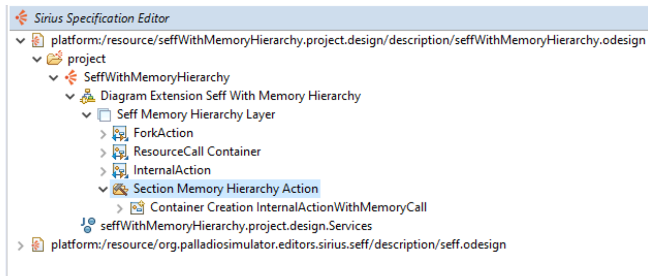


Figure A.25.: Screenshot of the .odesign File for the SeffWithMemoryHierarchy Viewpoint [Tru20]

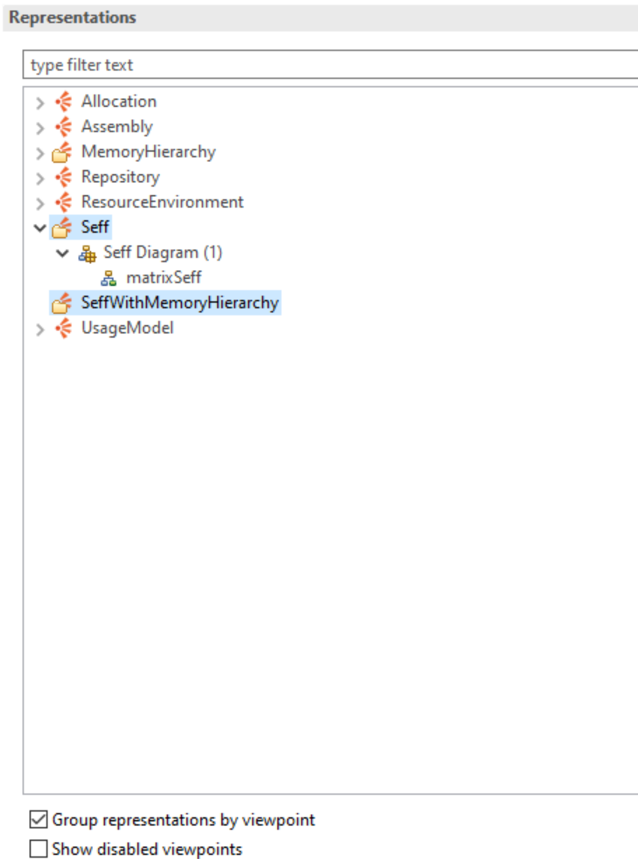


Figure A.26.: Screenshot of the Sirius Viewpoint Setting with the Viewpoints SEFF and SeffWithMemory-Hierarchy Activated [Tru20]

A.5.2. CPU and Memory Demand Calibration

To get the pure CPU demand (without memory hierarchy demand), we used the measurements we took from the sequential execution and the perf measurements. The intension in extracting the pure CPU demand is, that when considering the measurements from a sequential run, it contains both the CPU demands and the memory hierarchy demand. So, if we had used the measurements form a sequential run also for the multicore models, we would have also considered memory hierarchy demands. Thus, by modelling memory hierarchy demands explicitly—as we do in CB_3—and not using the pure CPU demands, we would have considered memory hierarchy demands twice.

To extracted the pure CPU demands from the sequential measurements, we use the perf measurements and calculate the demand by the following formula:

$$Demand_{CPU} = time_{singleThread} - time_{memoryHierarchy} \tag{A.1}$$

To estimate the $time_{memoryHierarchy}$ we use two different formulas: one for non-cache-line models and one for cache-line models.

A.5.2.1. Memory Time for Non-Cache-Line Models

For non-cache-line models, we assume the transfer of Java integers. Thus, we assume 4 bytes. We multiply the 4 bytes with the measured cache access times (load-operations) from perf and divide it by the memory bandwidth. The formula is the following:

$$time_{memoryHierarchy} = \left(\frac{load_{dcache} \times 4}{bandwidth_{L1}} + \frac{load_{L2} \times 4}{bandwidth_{L2}} + \frac{load_{L3} \times 4}{bandwidth_{L3}} + \frac{load_{DRAM} \times 4}{bandwidth_{DRAM}}\right) \tag{A.2}$$

or:

$$time_{memoryHierarchy} = 4 \times \left(\frac{load_{dcache}}{bandwidth_{L1}} + \frac{load_{L2}}{bandwidth_{L2}} + \frac{load_{L3}}{bandwidth_{L3}} + \frac{load_{DRAM}}{bandwidth_{DRAM}} \right) \tag{A.3}$$

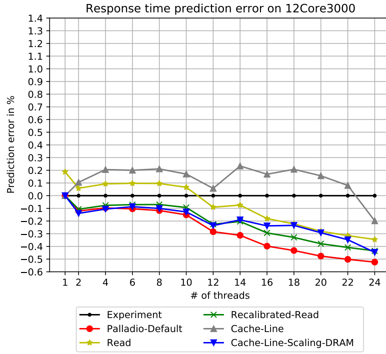
A.5.2.2. Memory Time for Cache-Line Models

In case we consider cache-line models, we do not multiply with 4 bytes but use the cache-line size. Only for the data transfer between the CPU registers and the L1 cache we assume a lower data-rate of the actual values (i.e., 4 bytes integer). In all the hardware systems we consider the cache-line size is 64 bytes.

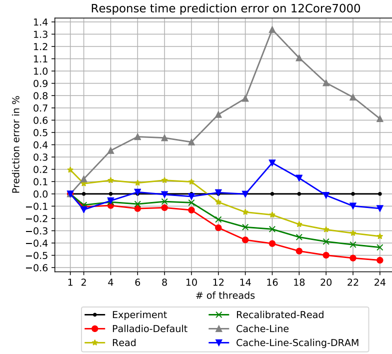
Thus, the following formula is used:

$$time_{memoryHierarchy} = \frac{load_{dcache} \times 4}{bandwidth_{L1}} + size_{cacheLine} \times \left(\frac{load_{L2}}{bandwidth_{L2}} + \frac{load_{L3}}{bandwidth_{L3}} + \frac{load_{DRAM}}{bandwidth_{DRAM}} \right) \tag{A.4}$$

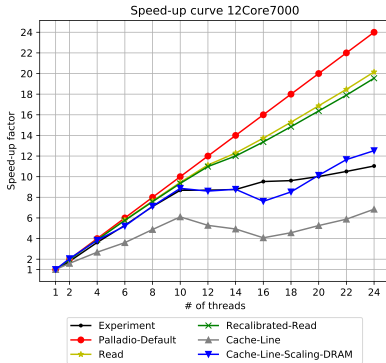
A.5.3. Results HPI Small (12 Cores)



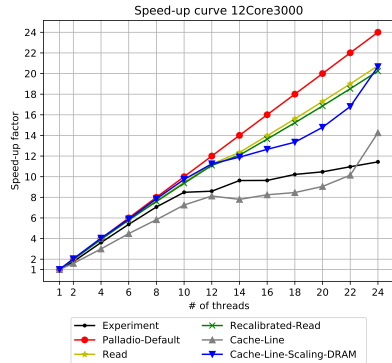
(a) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Small Use Case



(b) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Large Use Case

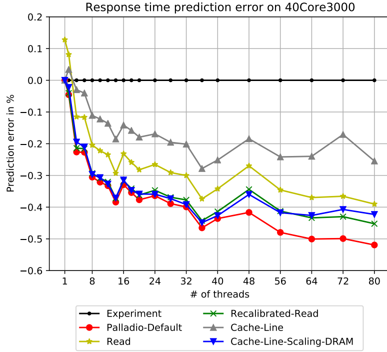


(c) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Small Use Case

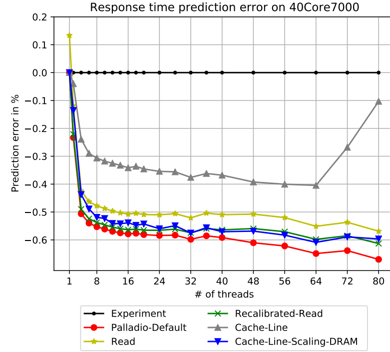


(d) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Large Use Case

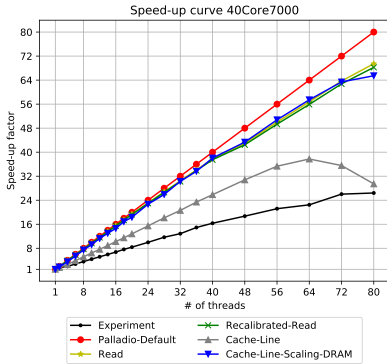
A.5.4. Results HPI Large (40 Cores)



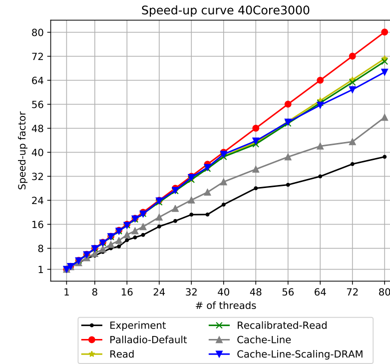
(a) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Small Use Case



(b) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Large Use Case

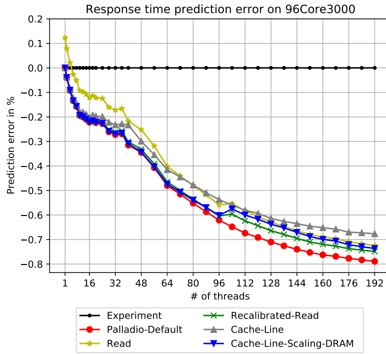


(c) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Small Use Case

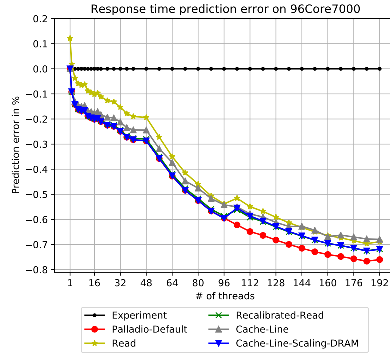


(d) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Large Use Case

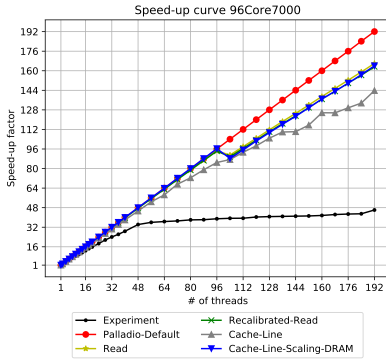
A.5.5. Results Stuttgart (96 Cores)



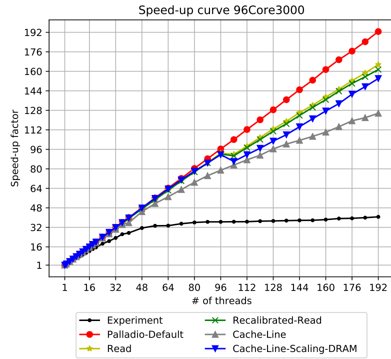
(a) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Small Use Case



(b) Comparison of Prediction Models: Prediction Error in % for the 12-Core Machine and Large Use Case



(c) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Small Use Case



(d) Comparison of Prediction Models: Speedup Diagram for the 12-Core Machine and Large use Case

A.6. CPU Simulator

A.6.1. Extension Points to Connect Trace-drive CPU Simulators to Palladio

A.6.1.1. SimCom Extension Point A

Listing A.7: de.uka.ipd.sdq.simucomframework.resources.ScheduledResource – getScheduledResource()

```
private IActiveResource getScheduledResource(final SimuComModel simuComModel,
    final String sensorDescription) {

  IActiveResource scheduledResource = null;
  // active resources scheduled by standard scheduling techniques
  if (getSchedulingStrategyID().equals(SchedulingStrategy.FCFS)) ||
    (getSchedulingStrategyID().equals(SchedulingStrategy.PROCESSOR_SHARING)) ||
    (getSchedulingStrategyID().equals(SchedulingStrategy.DELAY)) {
      ...
    } else {
      scheduledResource = getModel().getSchedulingFactory().createResourceFromExtension(
          getSchedulingStrategyID(), getNextResourceId(), getNumberOfInstances());
    }

    if (scheduledResource instanceof SimuComExtensionResource) {
      // The resource takes additional configuration that is available in the SimuComModel object
      // As the scheduler project is currently SimuCom-agnostic, we use the
      // SimuComExtensionResource class to initialize the resource wit a SimuCom-related object.
      ((SimuComExtensionResource) scheduledResource).initialize(simuComModel);
    }
    return scheduledResource;
  }
```

A.6.1.2. SimuCom Extension Point B

Listing A.8: de.uka.ipd.sdq.simucomframework.ExperimentRunner – run()

```
1  public static double run(SimuComModel model, long simTime) {
2        // ...
3        setupStopConditions(model);
4
5        // measure elapsed time for the simulation
6        double startTime = System.nanoTime();
7
8        ISimulationControl simulationControl = model.getSimulationControl();
9        simulationControl.start();
10
11       return System.nanoTime() - startTime;
12    }
```

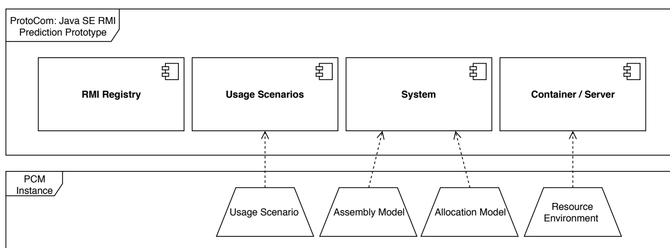


Figure A.30.: PCM influence on the SE RMI Prediction Prototpye [Gra18]

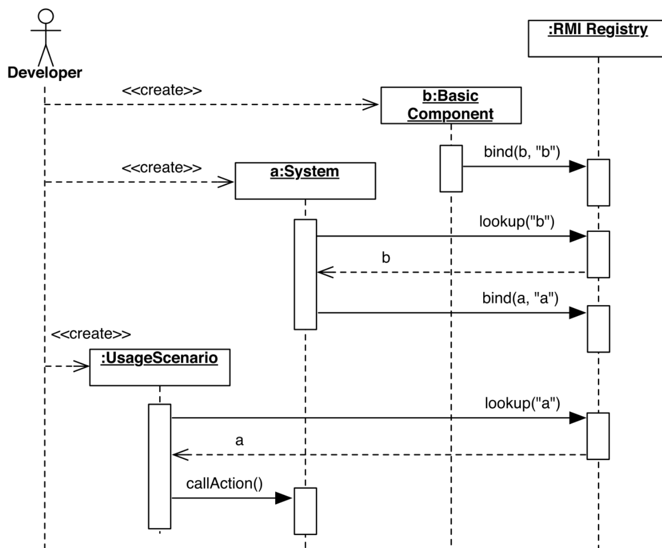


Figure A.31.: Sequence Diagram for Initialisation and Assembly using RMI [Gra18]

A.6.2. SimulatorBuilder Class

A.6.3. MaxSim Config File

Listing A.9: MaxSim: Hardware Configuration – 8Cores[Gra18]

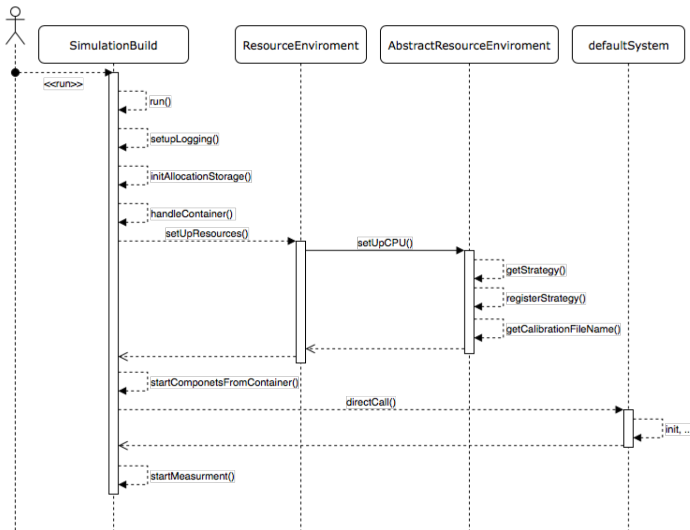


Figure A.32.: Sequence Diagram for Prototype without RMI [Gra18]

```
sim = {
  maxTotalInstrs = 1000000000000L;
  phaseLength = 10000;
  statsPhaseInterval = 10000;
  pointerTagging = true;
  ffReinstrument = true;
  logToFile = true;
};

sys = {
  caches = {
    l1d = {
      array = {
        type = "SetAssoc";
        ways = 8;
      };
      caches = 8;
      latency = 4;
      size = 32768;
    };

    l1i = {
      array = {
        type = "SetAssoc";
        ways = 4;
```

```
        };
        caches = 8;
        latency = 3;
        size = 32768;
      };

      l2 = {
        array = {
          type = "SetAssoc";
          ways = 8;
        };
        caches = 8;
        latency = 6;
        children = "l1i|l1d";
        size = 262144;
        MAProfCacheGroupId = 0;
      };

      l3 = {
        array = {
          hash = "H3";
          type = "SetAssoc";
          ways = 16;
        };
        banks = 8;
        caches = 1;
        latency = 30;
        children = "l2";
        size = 33554432;
        MAProfCacheGroupId = 1;
      };

      MAProfCacheGroupNames = "l2|l3";
    };

  cores = {
      haswell = {
        cores = 16;
        dcache = "l1d";
        icache = "l1i";
        type = "OOO";
      };
  };
  [...]
  l3 = {
        banks = 16;
        caches = 1;
        latency = 30;
        children = "l2";
        size = 67108864;
  };
```

A.6.4. ProtoCom Calibration

Listing A.10: MaxSim: Calibration Run-Config [Gra18]

```
process0 = {
  command = "./maxine/com.oracle.max.vm.native/generated/linux/maxvm \
  -XX:+MaxSimExitFFOnVMEnter \
  -XX:+MaxSimEnterFFOnVMExit \
  -XX:+MaxSimProfiling \
  -XX:+MaxSimPrintProfileOnVMExit \
  -cp /usr/local/src/calibrationTool.jar
        me.graef.sebastian.bachelor.thesis.Main";
  startFastForwarded = true;
  syncedFastForward = "Never";
};
```

Listing A.11: MaxSim: Calibration Results [Gra18]

```
# zsim stats
===
root: # Stats
 contention: # Contention simulation stats
  domain-0: # Domain stats
   time: 25707115262 # Weave simulation time
 time: # Simulator time breakdown
  init: 5369536005
  bound: 8900122799609
  weave: 1629998320947
  ff: 2072018500
[...]
 phase: 5500137 # Simulated phases
 haswell: # Core stats
  haswell-0: # Core stats
   cycles: 55001375142 # Simulated unhalted cycles
   [...]
  haswell-1: # Core stats
   cycles: 0 # Simulated unhalted cycles
   cCycles: 0 # Cycles due to contention stalls
[...]
```

A.7. Research Questions and Answers

Due to the fact, that the research question map to the contributions, we already discussed each research question in the corresponding chapter of the contribution. For the sake of better overview, we briefly summarise the outcome and answer to each research question in the following again.

A.7.0.1. RQ_1: Modelling of parallel performance relevant behaviour in massive parallel environments

$RQ_{1.1}$:**Are software architects able to model even simple parallel concepts of highly parallel systems in an efficient way?**

Answer: *We could show during an empirical user study using a controlled experiment, that current state of the art tool do not support SA in en efficient way.*

$RQ_{1.2}$: **Are software architects able to model the parallel software behaviour of an application with the help of current modelling languages, so that (a) the relevant performance characteristics are captured and expressed, and (b) all necessary information for performance evaluation is covered?**

Answer: *SA are currently not able to model (a) all relevant characteristics of parallel software, which results in (b) inaccurate performance predictions for parallel software in multicore enviorments.*

$RQ_{1.3}$: **How can software architects be supported by the task to create accurate performance perdition models efficiently?**

Answer: *By the help of a parallel AT catalogue SAs can be supported to create performance prediction models faster and with a higher user*

acceptance (usability). Further they can use the concept of overhead modelling to increase the accuracy of the predictions.

A.7.0.2. RQ_2: Performance behaviour of highly parallel applications in massive parallel environments:

$RQ_{2.1}$: **How do highly parallel applications behave in massive parallel environments (multicore systems) regarding response time (speedup), memory access rates (L1, L2, L3, RAM usage), and memory bandwidth utilisation?**

Answer: *In over 800 experiments we took 70,000 measurements. Thereby, we monitored the response time and memory accesses of the systems. Using these measurements we extracted the twelve performance curves given in Table 7.3 to describe the behaviour.*

$RQ_{2.2}$: **What factors influence performance the most in highly parallel applications?**

Answer: *In Table 7.1 we listed the top eight performance-influencing factors we identified by a structured literature reviews, expert interviews, and the experiments.*

$RQ_{2.3}$: **Does the choice of parallelisation strategy have a significant impact on behaviour?**

Answer: *The experiments show slight differences in the performance of the individual parallelisation paradigms. However, these differences are not signification for all thread-based paradigms. The only paradigm that diverges is the AKKA Actors implementation. Here we assume issues in the coding of the framework.*

$RQ_{2.4}$: **Do highly parallel applications show similar behaviour, which can be described by one or multiple performance curves?**

Answer: *In Table 7.3 we present performance curves for all the research resource demands. We used linear regression to extract the curves form the measurements. Thus, the curves describe the average behaviour for each demand type on all the tested machines.*

Finally, we can verify or falsify our hypothesis as follows:

$H_{2.1}$: **The speedup and performance behaviour of highly parallel applications depends heavily on the chosen parallelisation strategy or paradigm.**

Reject: *The chose of the parallelisation strategy does not have a high impact on the behaviour*

$H_{2.2}$: **The hardware architecture (e.g., number of CPU cores, memory bandwidth, memory hierarchies) of the execution environment has a strong impact on the performance of the parallel applications.**

Accept: *We measured differences in the normalised speedup for all the machines. Thus, they can verify that the hardware architecture has an impact on the performance. The biggest noticeable difference is between virtualised hardware and dedicated systems. Virtualised hardware show worse performance.*

$H_{2.3}$: **The speedup of a parallel application is not only influenced by the number of cores available in a system but also by additional hardware specific performance-influencing factors.**

Accept: *In Table 7.1 we listed the top eight performance-influencing factors we identified*

A.7.0.3. RQ_3: Performance Prediction Models

$RQ_{3.1}$: **Are current simulation-based performance prediction approaches capable of predicting the performance of parallel and highly parallel systems accurately?**

Answer: *The experiments we performed in [FH16; FSH17] show that current state of the art performance prediction approaches are up to 80% off when trying to predict the response-time for parallel applications in multicore environments*

$RQ_{3.2}$:**If not, what are the missing characteristics of software behaviour that must be included in performance prediction models (performance-influencing factors)?**

Answer: *Table 7.1 shows the top eight most performance-influencing factors, we gained from a structured literature reviews, expert interviews, and experimenting.*

$RQ_{3.3}$: **Can modelling the additional performance-influencing factors improve the overall accuracy of performance prediction?**

Answer: *We showed that booths, the use of performance curves, which are an abstract representation of the PPiFs, and the modelling of memory hierarchies help to improve the performance predictions for parallel applications in multicore environments. Thereby we achieve an accuracy up to 89% for certain scenarios. That result is by 57% more accurate than the pure Palladio approach.*

A.7.0.4. RQ_4: CPU Simulators

$RQ_{4.1}$: **Can CPU Simulators be used by software architects to evaluate the response time of parallel architectural designs?**

Answer: *We were able to show, that it is possible to transform the architectural models into a performance prototype. Which we again can use as input for multicore CPU simulators to determine the response or execution time of a parallel application.*

$RQ_{4.2}$: **How would the integration of CPU simulators alter the process of performance predictions?**

Answer: *In Section 9.3 we sketched two approaches to include CPU simulators into the performance prediction workflow: (1) a trace-driven approach, (2) a source code-driven approach. In both cases we use the PCM without additional informations as starting point. Next, we transform the PCM by the use of solvers either into a trace-file or a performance prototype, which we finally use as input for the multicore simulators.*

$RQ_{4.3}$: **Does the use of CPU Simulators increase the performance prediction accuracy for parallel applications in multicore environments?**

Answer: *We implemented the source code-driven approach to evaluate the accuracy of the performance prediction using multicore CPU simulators. Thereby, we used a complex use case example the Bank Transaction Example (see Sec. 5.2.1). The prediction accuracy of this approach for the given example was with an accuracy from* 2.50% *to* 15.29% *very inaccurate and up to* 54% *worse than the pure Palladio approach.*

Therefore, we have to reject our hypothesis H_4*: CPU simulators—used in other domains (e.g, hardware vendors)—can help to improve the predictions for parallel applications on multicore CPUs.*

List of Figures

1.1. Overview of the solution and contributions presented in this thesis 8

2.1. Concurrent vs. Parallel Execution 12
2.2. Abstract overview of threads and stream processing 15
2.3. Example of an Actor System 19
2.4. Example of SISD 20
2.5. Example of SIMD 21
2.6. Example of MIMD 21
2.7. Exemplification of SMP 21
2.8. Exemplification of NUMA 22
2.9. Exemplification of a Distributed Memory Architecture 23
2.10. Example of a Common Hierarchical Multicore Processor 24
2.11. Example Stream Application 28
2.12. Categorisation of Parallel Patterns 29
2.13. Simulation of target components 30
2.14. Overview of the PCM 35
2.15. Overview of the SimuCom Solver 38
2.16. Detailed View of SimuCom 38
2.17. Overview of the ProtoCom M2C Transformation 41
2.18. Architectural View of JavaSE Performance Prototype 42
2.19. Process of the AT Application Process 43
2.20. Mapping of Foundations to Contributions 47

3.1. Applied Design Science Framework to Archive the *RG* 51
3.2. Research Process of this Thesis 53

4.1. Overview of the Systematic Literature Review Process 62
4.2. Filtering Process 69

5.1. Domain View of the Bank Transaction Example 92

6.1. Overview of the Research Method for Contribution C_1 100
6.2. SEFF Definition of the Sequential Model 104
6.3. Stepwise extension of loop to a parallel loop 115
6.4. SEFF Representation of the Unfolded the Parallel Loop Example 116
6.5. AT Profile for Parallel Loop Extension 118
6.6. Categorisation of Parallel Patterns 129
6.7. Mapping of PCM2QPN: (a) LoopAction, (b) Asynchronous Fork, (c) Synchronous Fork 131
6.8. Mapping PCM2QPN: (a) asynchronous parallel loop, (b) synchronous parallel loop 134
6.9. Goals, Hypotheses, Questions, and Metrics of the User Study . . 140
6.10. Overview of the User Study 142
6.11. Liker Plots of Questions Five to Seven 147

7.1. Overview of the Research Method for Contribution CB_2 154
7.2. Measurements of Speedup Functions for Different Resource Demands on a 40-Core System with Enabled Hyper-threading . 156
7.3. Overview of Experiment Setup using ProtoCom as Resource Demand Factory 162
7.4. Speedup for Different Parallelisation Paradigms 167
7.5. Cache Miss Rate for Java Threads on the Server in Stuttgart . . 169
7.6. Cache Accesses for Java Threads on the Server in Stuttgart . . . 170
7.7. Comparison of the Four Hardware Environments Exemplified by Using Java Threads, Mandel Set, and Count Number Demand 172
7.8. Comparison of the Four Hardware Environments Using Java Threads and the Mandel Set Demand 179
7.9. Profile Example for Parallel For-loop 180
7.10. Property View of the Applied Parallel Loop AT 181
7.11. Speedup Curves for the Applications from the OMP2012 Benchmark Suite 184

8.1. Overview of the Research Method for Contribution CB_3 194
8.2. Overview of the Profile Extension of the ResourceContainer . . 202
8.3. Meta-Model Extension Containing the New Elements for the Memory Hierarchy 204
8.4. Overview of the Evaluation Process for CB_3 210

8.5. Repository Model for the Matrix Multiplication Use Case 212
8.6. SEFF Model for the Matrix Multiplication Use Case with Two Threads. 214
8.7. Repository Model for the Matrix Multiplication Use Case 215
8.8. Comparison of Prediction Models: Prediction Error in % for all Machines and Use Cases 220

9.1. Overview of the Research Method for Contribution CB_4 228
9.2. Overview of Multicore CPU Simulators 231
9.3. Tejas Feature Net 232
9.4. Sniper Feature Net 234
9.5. zsim Feature Net 234
9.6. MaxSim Feature Net 235
9.7. Gem5 Feature Net 235
9.8. MARSSx86 Feature Net 236
9.9. MultiSim Feature Net 237
9.10. Inclusion Strategy Using SimuCom and Trace-driven Multicore CPU Simulators 241
9.11. Inclusion Strategy Using ProtoCom and Source Code-base Multicore CPU Simulators 243
9.12. Chart based visualisation of the measurements for small and large use cases 248

10.1. Prediction Error for the Combined Approach: Matrix Multiplication Performance Curves and Cache-line Memory Model 258

A.1. Publication Overview 271
A.2. Speedup for Threads and OpenMP 286
A.3. Speedup for Streams and Actors 287
A.4. Speedup for Threads and OpenMP 288
A.5. Speedup for Streams and Actors 289
A.6. Speedup for Threads and OpenMP 290
A.7. Speedup for Streams and AKKA Actors 291
A.8. L2 Cache Behaviour 292
A.9. L3 Cache Behaviour 293
A.10. L2 cache behaviour for Threads and OpenMP 294
A.11. L2 Cache Behaviour for Streams and Actors 295

A.12. L3 Cache Behaviour for Threads and OpenMP 296
A.13. L3 Cache Behaviour for Streams and AKKA Actors 297
A.14. L2 Cache Behaviour for Threads and OpenMP 298
A.15. L2 Cache Behaviour for Streams and AKKA Actors 299
A.16. L3 cache behaviour Threads and Streams 300
A.17. L3 cache behaviour for Streams and AKKA Actors 301
A.18. L3 Cache Behaviour for Threads and OpenMP 302
A.19. L3 Cache Behaviour for Streams and AKKA Actors 303
A.22. Screenshoot of the .odesign File for the Memory Hierarchy . . . 307
A.23. Screenshot of the Memory Hierarchy Editor with Palette Showing Elements That Can Be Added to the Diagram 308
A.24. Screenshot of the Memory Hierarchy Editor with an Edit Dialog 309
A.25. Screenshot of the .odesign File for the SeffWithMemoryHierarchy Viewpoint 309
A.26. Screenshot of the Sirius Viewpoint Setting with the Viewpoints SEFF and SeffWithMemory-Hierarchy Activated 310
A.30. PCM influence on the SE RMI Prediction Prototpye 318
A.31. Sequence Diagram for Initialisation and Assembly using RMI . 318
A.32. Sequence Diagram for Prototype without RMI 319

List of Tables

1.1. Overview of the thesis structure 10

2.1. Comparison of Concurrency and Parallelism 13

4.1. Characteristics Used for Categorising 67
4.2. Classification of Sources for General Software Engineering 70
4.3. Classification of sources for HPC, Embedded Systems, and SPE . . 71

5.1. Summary of Resource Demand Characteristics 91

6.1. Simulations and Measurements Summary 106
6.2. Summary for Different Extension Strategies 111
6.3. Simulations and Measurements Summary Using a Parallal-Loop-Action 120
6.4. Summary of the User Study 145

7.1. Prioritised list of PPiFs after ranking by experts 161
7.2. Overview of the hardware environments and their configuration 164
7.3. Extracted Performance Curves for Dedicated Machines Based on the Speedup Behaviour of the Demands 178
7.4. Mapping of benchmark applications to expected demand types 183
7.5. Shows the inaccuracy of the pure Palladio approach in comparison to the Palladio + Performance Curve approach for the SPEC OMP2012 Benchmark set. 187
7.6. Shows the accuracy gain of the performance curve approach in comparison to the pure Palladio approach 187

8.1. Mean Prediction Error for the Different Use Cases and Modelling Approaches 219

9.1. Overview and Compression of CPU Multicore Simulators 240

9.2. Overview of all measurements and simulation results from SimuCom and the MaxSim approach 247

10.1. Comparision of Cache-Line and Cache-Line with Performance Curvees 258
10.2. Summary of working combinations 260

A.1. Extracted Performance Curves for Dedicated Machines based on the Speedup Behaviour of the Demands 304
A.2. Extracted Performance Curves for Virtualised Machines Based on the Speedup Behaviour of the Demands 304

List of Abbreviations

AT Architectural Template

BIS Business Information System

CB Contribution

ccNUMA cache-coherent nonuniform memory access systems

CGSPN Coloured Generalised Stochastic Petri Net

CPN Coloured Petri Net

CPU Central Processing Unit

DSL Domain Specific Language

ELF Executable and Linking Format

GSPN Generalised Stochastic Petri Net

GQM Goal-Question-Metric

HPC High Performance Computing

HQPN Hierarchical Queuing Petri Net

ISA Instruction Set Architecture

L1 Level 1 Cache

L2 Level 2 Cache

L3 Last Level Cache

LQN Layered Queuing Network

m2t model-to-text

m2m model-to-model

m2c model-to-code

MDSD Model-driven Software Development

MIMD Multiple Instruction and Multible Data

MISD Multiple Instruction and Single Data

MPI Message Passing Interface

NUMA Non-Uniform Memory Access

OOO Out-Of-Order

PCM Palladio Component Model

PN Petri Net

PPiFs Parallel Performance-influencing Factors

PPiF Parallel Performance-influencing Factor

QN Queuing Network

QoS Quality of Service

QPN Queuing Petri Net

RAM Random Access Memory

SA Software Architect

SEFF Service Effect Specification

SIMD Single Instruction and Multiple Data

SISD Single Instruction and Single Data

SLO Service-level Objective

SLR Systematic Literature Review

SMP Symmetric Multiprocessors

SPE Software Performance Engineering

SPN Stochastic Petri Net

UE Unit of Execution

ue Unit of Execution

Literature References

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. “A View of the Paral. Comp. Landscape”. In: *Comm. of the ACM* 10 (2009), pp. 56–67 (cit. on p. 70).

[ADKT17] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. “Fastflow: high-level and efficient streaming on multi-core”. In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017) (cit. on pp. 70, 74).

[AK99] G. A. Agha and W. Kim. “Actors: A Unifying Model for Paral. and Distr. Comp.” In: *Journal of Systems Architecture* 15 (1999), pp. 1263–1277.

[Ant19] I. Antunovic. “Simulation, Modeling and Verification Environment for Multicore Software Architectures for Automotive Embedded Systems”. Master theses Theses at the Reliable Software Systems Group at University Stuttgart. May 2019.

[AR06] S. Akhter and J. Roberts. *Multi-Core Programming*. Intel press Hillsboro, 2006 (cit. on p. 229).

[AS16] A. Akram and L. Sawalha. “A Comparison of x86 Computer Architecture Simulators”. In: *CEUR Workshop Proc.* 1691 (2016), pp. 21–27. arXiv: arXiv:1603.07016v1. URL: http://scholarworks.wmich.edu/casrl%7B%5C_%7Dreports (cit. on pp. 30–32, 227).

[BBE+11] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. R. Segovia, and C. Scordino. “Resource Management on Multicore Systems: The ACTORS Approach”. In: *IEEE Micro* 3 (2011), pp. 72–81 (cit. on pp. 71, 75).

[BBM13] M. Becker, S. Becker, and J. Meyer. “Simulizar: Design-time modeling and performance analysis of self-adaptive systems”. In: *Software Engineering 2013* (2013) (cit. on p. 39).

[BDH08] S. Becker, T. Dencker, and J. Happe. “Model-driven generation of performance prototypes”. In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, pp. 79–98 (cit. on pp. 162, 195).

[BDIS04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. “Model-based Performance Prediction in Software Development: A Survey”. In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310 (cit. on pp. 32, 33).

[Bea15] J. Beard. *Streaming And Dataflow*. `https://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html`. (Accessed on 02/03/2020). Sept. 2015 (cit. on p. 15).

[Bec08] S. Becker. “Coupled model transformations for QoS enabled component-based software design”. PhD thesis. Universität Oldenburg, 2008 (cit. on pp. 38, 39, 41, 42).

[Bec17] M. W. Becker. “Engineering Self-Adaptive Systems with Simulation-Based Performance Prediction”. PhD thesis. Universität Paderborn, 2017 (cit. on pp. 36, 39, 40).

[BGOS12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. “Accuracy evaluation of gem5 simulator system”. In: *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*. IEEE. 2012, pp. 1–7 (cit. on p. 235).

[BHS07] F. Buschmann, K. Henney, and D. Schimdt. *Pattern-oriented Software Architecture: on patterns and pattern language*. Vol. 5. John wiley & sons, 2007 (cit. on p. 26).

[BK02] F. Bause and P. S. Kritzinger. *Stochastic petri nets*. Vol. 1. Citeseer, 2002 (cit. on pp. 45, 46).

[BKBR11] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. “Architecture-based reliability prediction with the palladio component model”. In: *IEEE Transactions on Software Engineering* 38.6 (2011), pp. 1319–1339 (cit. on p. 33).

[BKL08] C. Bienia, S. Kumar, and K. Li. “PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors”. In: *2008 IEEE International Symposium on Workload Characterization*. IEEE. 2008, pp. 47–56 (cit. on p. 238).

[BKR09] S. Becker, H. Koziolek, and 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. 9, 33, 195).

[BSA+10] R. Brown, E. Shoop, J. Adams, C. Clifton, M. Gardner, M. Haupt, and P. Hinsbeeck. “Strategies for Preparing Computer Science Students for the Multicore World”. In: *Proceedings of the 2010 ITiCSE*. ACM. 2010, pp. 97–115 (cit. on pp. 70, 73).

[BVS13] R. Buyya, C. Vecchiola, and S. T. Selvi. “Chapter 7 - High-Throughput Computing: Task Programming”. In: *Mastering Cloud Computing*. Ed. by R. Buyya, C. Vecchiola, and S. T. Selvi. Boston: Morgan Kaufmann, 2013, pp. 211–252. URL: http://www.sciencedirect.com/science/article/pii/B9780124114548000073 (cit. on p. 17).

[CBM+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee. 2009, pp. 44–54.

[CGIP16] D. Cerotti, M. Gribaudo, M. Iacono, and P. Piazzolla. “Modeling and analysis of performances for concurrent multithread applications on multicore and graphics processing unit systems”. In: *Concurrency and Computation: Practice and Experience* 28.2 (2016), pp. 438–452 (cit. on pp. 71, 74, 77, 80).

[CHE11] T. E. Carlson, W. Heirmant, and L. Eeckhout. “Sniper: Exploring the level of abstraction for scalable and accurate parallel multicore simulation”. In: *2011 Int. Conf. High Perform. Comput. Networking, Storage Anal.* September (2011), pp. 1–12.

[Cli81] W. D. Clinger. “Foundations of actor semantics”. In: (1981) (cit. on p. 18).

[CR94] V. R. B. G. Caldiera and H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 139).

[CRB+11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and experience* 41.1 (2011), pp. 23–50 (cit. on p. 80).

[DE98] L. Dagum and R. Enon. "OpenMP: An Industry Standard API for Shared-Memory Programming". In: *Computational Science & Engi., IEEE* 1 (1998), pp. 46–55.

[Det20] S. Dettenmaier. "Integrating and Automating a Multicore CPU Simulator into Palladio". B.S. thesis. 2020 (cit. on pp. 228, 271).

[DG04] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150 (cit. on p. 125).

[DH84] A. Douady and J. H. Hubbard. *Etude dynamique des polynômes complexes. Partie I, volume 84 of Publications Mathématiques d'Orsay*. 1984 (cit. on p. 84).

[DMN12] J. Diaz, C. Munoz-Caro, and A. Nino. "A Survey of Paral. Programming Models and Tools in the Multi and Many-Core Era". In: *Paral. and Distr. Systems, IEEE Transactions on* 8 (2012), pp. 1369–1386 (cit. on pp. 71, 74, 77).

[Doy14] R. Doyle. *Using Akka and Scala to Render a Mandelbrot Set*. `https://blog.scottlogic.com/2014/08/15/using-akka-and-scala-to-render-a-mandelbrot-set.html`. (Accessed on 02/10/2020). Aug. 2014 (cit. on p. 19).

[DVAE19] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. "RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors". In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2019, pp. 257–267.

[EB16] R. Escobar and R. V. Boppana. "Performance prediction of parallel applications based on small-scale executions". In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE. 2016, pp. 362–371 (cit. on pp. 71, 75, 77, 80).

[EBS+12] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. “Dark Silicon and the End of Multicore Scaling”. In: *IEEE Micro* 3 (2012), pp. 122–134.

[EE10] S. Eyerman and L. Eeckhout. “Modeling Critical Sections in Amdahl’s Law and its Implications for Multicore Design”. In: *ACM SIGARCH Computer Architecture News*. 3. ACM. 2010, pp. 362–370 (cit. on p. 70).

[Eij17] V. Eijkhout. *Parallel Programming in MPI and OpenMP*. 2017 (cit. on pp. 17, 26, 28).

[Erb12] B. Erb. “Concurrent Programming for Scalable Web Architectures”. Diploma Thesis. Institute of Distributed Systems, Ulm University, Apr. 2012. URL: `http://www.benjamin-erb.de/thesis` (cit. on p. 26).

[FBKK19] M. Frank, S. Becker, A. Kaplan, and 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*. ACM. 2019, pp. 21–24 (cit. on pp. 84, 154, 158–160, 162, 195, 271).

[FH16] M. Frank and M. Hilbrich. “Performance Prediction for Multicore Environments—An Experiment Report”. In: *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany*. 2016. URL: `https://sdqweb.ipd.kit.edu/typo3/sdq/fileadmin/user_upload/palladio-conference/2016/papers/Performance_Prediction_for_Multicore_Environments_-_An_Experiment_Report.pdf` (cit. on pp. 7, 49, 52, 59, 61, 62, 81, 91, 100–103, 135, 151, 195, 211–213, 216, 249, 271, 325).

[FH18] M. Frank and A. Hakamian. “An Architectural Template for Parallel Loops and Sections”. In: *Proceedings of the Symposium on Software Performance 2018, 7-9 November 2018, Hildesheim, Germany*. 9th Symposium on Software Performance 2018. Hildesheim, Nov. 2018. URL: `https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2018/papers/FrankHakamian18.pdf` (cit. on pp. 130, 271).

[FHB20] M. Frank, A. Hakamian, and S. Becker. "Defining a Formal Semantic for Parallel Patterns in the Palladio Component Model Using Hierarchical Queuing Petri Nets". In: *Software Architecture*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, and U. Zdun. Cham: Springer International Publishing, 2020, pp. 381–394 (cit. on p. 100).

[FHL15] M. Frank, M. Hilbrich, and S. Lehrig. "Improved Scalability for Job-centric Monitoring in Distributed Infrastructures". In: *CGW Workshop 15 Proceedings*. ISBN 978-83-61433-14-9. ACC Cyfronet AGH, Oct. 2015, pp. 79–80.

[FHLB17] M. Frank, M. Hilbrich, S. Lehrig, and 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 pp. 7, 61, 64, 65, 67, 71, 73–76, 78, 271).

[FKB18] M. Frank, F. Klinaku, and 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 (cit. on pp. 63, 76, 271).

[FKHB19] M. Frank, F. Klinaku, M. Hilbrich, and 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 pp. 100, 106, 107, 271).

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960 (cit. on pp. 19, 20).

[FSH17] M. Frank, S. Staude, and M. Hilbrich. "Is the PCM Ready for ACTORs and Multicore CPUs? - A Use Case-based Evaluation". In: *Proceedings of the Symposium on Software Performance 2017, 9-10 November 2017, Karlsruhe, Germany*. 8th Symposium on Software Performance 2017. Karlsruhe, Nov. 2017. URL: `http://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2017/papers/Is_the_PCM_Ready_for_ACTORs_and_Multicore_CPUs_A_Use_Case-based_`

Evaluation.pdf (cit. on pp. 7, 49, 52, 55, 59, 61, 62, 151, 245, 246, 249, 271, 325).

[FSK+20] M. Frank, L. Schmid, A. Kaplan, L. Greiner, A. Koziolek, and S. Becker. "Performance Curves for better Performance Predictions of Parallel Applications in Multicore Environments". In: *Companion of the 2020 ACM/SPEC International Conference on Performance Engineering*. currently under review. ACM. 2020 (cit. on p. 154).

[FV04] L. Fuentes-Fernández and A. Vallecillo-Moreno. "An introduction to UML profiles". In: *UML and Model Engineering* 2.6-13 (2004), p. 72 (cit. on p. 201).

[GA12] I. Gray and N. C. Audsley. "Challenges in Softw. Development for Multicore System-on-Chip Development". In: *Rapid System Prototyping (RSP), 2012 23rd IEEE Int. Sym. on*. IEEE. 2012, pp. 115–121 (cit. on pp. 71, 75).

[GEE10] D. Genbrugge, S. Eyerman, and L. Eeckhout. "Interval simulation: Raising the level of abstraction in architectural simulation". In: *HPCA - 16 2010 Sixt. Int. Symp. High-Performance Comput. Archit.* (2010), pp. 1–12. URL: http://ieeexplore.ieee.org/document/5416636/ (cit. on p. 31).

[GF19] P. Gruber and M. Frank. "Modelling and Predicting Memory Behaviour in Parallel Systems with Network Links—Palladio-based Experiment Report". In: *Proceedings of the Symposium on Software Performance 2019, 4-6 November 2019, Wuerzburg, Germany*. 10th Symposium on Software Performance 2019. Wuerzburg, Nov. 2019. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2019/Papers/SSP2019_paper_3.pdf (cit. on pp. 194–196, 271).

[GHK+13] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. "Chapter 1 - Introduction to Parallel Programming". In: *Heterogeneous Computing with OpenCL (Second Edition)*. Ed. by B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. Second Edition. Boston: Morgan Kaufmann, 2013, pp. 1–13. URL: http://www.sciencedirect.com/science/article/pii/B9780124058941000012 (cit. on pp. 17, 18).

[GR04] J. Gummaraju and M. Rosenblum. "Stream processing in general-purpose processors". In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 04'), Boston, MA, USA*. 2004, pp. 9–13 (cit. on p. 15).

[Gra18] S. Graef. *Connecting Palladio with multicore CPU simulators*. Englisch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Sichere und Zuverlässige Softwaresysteme. Bachelorarbeit. Oct. 2018. URL: `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-2018-70&engl=0` (cit. on pp. 30, 228, 231, 232, 234–237, 240, 241, 243, 271, 318, 319, 321).

[Gra69] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM journal on Applied Mathematics* 2 (1969), pp. 416–429.

[Gre19] L. Greiner. "Bewertung verschiedener Parallelisierungsstrategien im Hinblick auf Leistungsfähigkeit von paralleler Programmausführung". Bachelor Theses at the Chair for Software Design and Quality in Karlsruhe in collaboration with the Reliable Software Systems Group at University of Stuttgart. 2019 (cit. on pp. 158, 165, 167, 169, 170, 271, 286–303).

[Gru19] P. Gruber. "Using Palladio network links to model multicore architecture memory hierarchies". B.S. thesis. 2019 (cit. on pp. 194, 200, 212, 271).

[Gru20] P. Gruber. "Benchmarking and Measuring of Performance-influencing Factors". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2020 (cit. on pp. 93, 271).

[GS13] N. Giacaman and 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. 211).

[Hap08] J. Happe. "Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments". Dissertation. University of Oldenburg, Germany, Aug. 2008. URL: `http://`

oops.uni-oldenburg.de/827/1/happre08.pdf (cit. on pp. 33, 50, 59, 80, 114, 158, 161, 196, 211, 213, 221).

[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 p. 206).

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence.* IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245 (cit. on pp. 17, 125).

[HC10] A. Hevner and S. Chatterjee. "Design science research in information systems". In: *Design research in information systems.* Springer, 2010, pp. 9–22 (cit. on pp. 50, 51, 58).

[HF15a] M. Hilbrich and M. Frank. "Analysis of Series of Measurements from Job-Centric Monitoring by Statistical Functions". In: *CGW Workshop 15 Proceedings.* ISBN 978-83-61433-14-9. ACC Cyfronet AGH, Oct. 2015, pp. 81–82.

[HF15b] M. Hilbrich and M. Frank. *Better Resource Usage by Job-Centric Monitoring.* Symposium on Software Performance 2015, Poster, Munich, Germany. Nov. 2015.

[HF17a] M. Hilbrich and M. Frank. "Analysis of Series of Measurements from Job-Centric Monitoring by Statistical Functions". In: *Computer Science* 18.1 (2017), p. 2. URL: https://journals.agh.edu.pl/csci/article/view/1791.

[HF17b] M. Hilbrich and M. Frank. "Debugging a Complex Systems, the Long Way from Data to Knowledge". In: *Proceedings of the Symposium on Software Performance 2017, 9-10 November 2017, Karlsruhe, Germany.* 8th Symposium on Software Performance 2017. Karlsruhe, Nov. 2017. URL: http://www.performance-symposium.org/2017/program/debugging-a-complex-systems-the-long-way-from-data-to-knowledge/.

[HF17c] M. Hilbrich and M. Frank. "Enforcing Security and Privacy via a Cooperation of Security Experts and Software Engineers-a Model-based Vision". In: *Proceedings of the 7th IEEE International Symposium on Cloud and Service Computing, 22-25 November 2017, Kanazawa, Japan.* 7th IEEE International Symposium on Cloud and Service Computing. Kanazawa, Nov. 2017.

[HF17d] M. Hilbrich and M. Frank. "Time-Aligned Similarity Calculations for Job-Centric Monitoring". In: *2017 International Conference on Green Informatics (ICGI).* IEEE. 2017, pp. 229–237.

[HF18a] M. Hilbrich and M. Frank. "Abstract Fog in the Bottle-Trends of Computing in History and Future". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE. 2018, pp. 519–522.

[HF18b] M. Hilbrich and M. Frank. "HPC and SPE Need to Learn from Each Other-Knowledgetransformation Patterns". In: *The 18th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations.* SCAN2018 Book of Abstracts. Tokyo, Japan, 2018, pp. 58–59.

[HFL16] M. Hilbrich, M. Frank, and S. Lehrig. "Security Modeling with Palladio—Different Approaches". In: *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany.* 2016. URL: `https://sdqweb.ipd.kit.edu/typo3/sdq/fileadmin/user_upload/palladio-conference/2016/papers/Security_Modeling_with_Palladio-Different_Approaches.pdf` (cit. on p. 33).

[HKM08] W.-m. Hwu, K. Keutzer, and T. G. Mattson. "The Concurrency Challenge". In: *IEEE Design & Test of Computers* 4 (2008), pp. 312–320 (cit. on pp. 70, 72).

[HL04] A. Hofstein and V. N. Lunetta. "The Laboratory in Science Education: Foundations for the Twenty-First Century". In: *Science education* 1 (2004), pp. 28–54.

[HL08] S. Hoffmann and R. Lienhart. *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++.* Springer-Verlag, 2008 (cit. on p. 16).

[HLF16] M. Hilbrich, S. Lehrig, and M. Frank. *Measured Values Lost in Time—or How I rose from a User to a Developer of Palladio.* `http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-213813`. 2016.

[HM08] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 7 (2008), pp. 33–38 (cit. on p. 110).

[HM93] M. Herlihy and J. E. B. Moss. *Transactional Memory: Architectural Support for Lock-free Data Structures.* 2. ACM, 1993.

[HO09] P. Haller and M. Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theoretical Computer Science* 2 (2009), pp. 202–220 (cit. on pp. 70, 72, 76).

[HPD09] P. E. Hadjidoukas, G. C. Philos, and V. Dimakopoulos. "Exploiting fine-grain thread Paral.ism on multicore architectures". In: *Scientific Programming* 4 (2009), pp. 309–323 (cit. on pp. 71, 74, 77).

[IBM18] IBM. *IBM Knowledge Center, Performance tuning tools and methodology.* 2018 (cit. on p. 94).

[IDSM05] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. "An approach to performance prediction for parallel applications". In: *European Conference on Parallel Processing*. Springer. 2005, pp. 196–205 (cit. on pp. 70, 74).

[ISC+15] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf. "How Many Threads will be too Many? On the Scalability of OpenMP Implementations". In: *Euro-Par 2015: Paral. Processing*. Springer, 2015, pp. 451–463 (cit. on pp. 70, 72).

[JBC+14] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, et al. "SPEC ACCEL: A standard application suite for measuring hardware accelerator performance". In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer. 2014, pp. 46–67.

[Jen13] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use.* Vol. 1. Springer Science & Business Media, 2013 (cit. on p. 45).

[JPH+09] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. "Shore-MT: A Scalable Storage Manager for the Multicore Era". In: *Proceedings of the 12th Int. Conf. on Extending Database Technology: Advances in Database Technology.* ACM. 2009, pp. 24–35.

[JS09] S. P. Jones and S. Singh. "A Tutorial on Paral. and Concurrent Programming in Haskell". In: *Advanced Functional Programming.* Springer, 2009, pp. 267–305.

[KBAW94] R. Kazman, L. Bass, G. Abowd, and M. Webb. "SAAM: A method for analyzing the properties of software architectures". In: *Proceedings of 16th International Conference on Software Engineering.* IEEE. 1994, pp. 81–90 (cit. on p. 3).

[KBB+09] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. "Systematic Literature Reviews in Softw. Engi.–A Systematic Literature Review". In: *Information and Softw. technology* 1 (2009), pp. 7–15 (cit. on p. 61).

[KBH+14] H. Koziolek, S. Becker, J. Happe, P. Tuma, and T. de Gooijer. "Towards Softw. Perf. Engi. for Multicore and Manycore Systems". In: *ACM SIGMETRICS Perf. Evaluation Review* 3 (2014), pp. 2–11.

[KC07] B. Kitchenham and S. Charters. *Guidelines for performing Systematic Literature Reviews in Softw. Engi.* 2007 (cit. on pp. 62, 63).

[KDJ04] B. A. Kitchenham, T. Dyba, and M. Jorgensen. "Evidence-based Softw. Engi." In: *Proceedings of the 26th Int. Conf. on Softw. Engi.* IEEE Computer Society. 2004, pp. 273–281 (cit. on p. 61).

[KFB18] F. Klinaku, M. Frank, and S. Becker. "CAUS: An Elasticity Controller for a Containerized Microservice". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* ACM. 2018, pp. 93–98.

[KHF19] F. Klinaku, A. Hakamian, and M. Frank. “A Process Model for Elastic and Resilient IoT Applications with Emergent Behaviors”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* IEEE. 2019, pp. 27–30.

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. “The architecture tradeoff analysis method”. In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193).* IEEE. 1998, pp. 68–78 (cit. on p. 3).

[KL14] C. Klaussner and S. Lehrig. “Extensible Performance Prototype Transformations for Multiple Platforms”. In: *Bachelor thesis, Software Engineering Group, University of Paderborn, Software Engineering Group, Paderborn, Germany* (2014) (cit. on pp. 40–42).

[KMZS08] H. Kasim, V. March, R. Zhang, and S. See. “Survey on Paral. Programming Model”. In: *Net. and Paral. Comp.* Springer, 2008, pp. 266–275.

[Knu97] D. E. Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997 (cit. on p. 84).

[Koz08] H. Koziolek. “Parameter dependencies for reusable performance specifications of software components”. PhD thesis. Universität Oldenburg, 2008 (cit. on pp. 45, 130–132, 135).

[Koz10] H. Koziolek. “Performance Evaluation of Component-based Softw. Systems: A Survey”. In: *Perf. Evaluation* 8 (2010). Special Issue on Softw. and Perf., pp. 634–658.

[KP11] T. Karcher and V. Pankratius. “Run-time Automatic Perf. Tuning for Multicore Applications”. In: *Euro-Par 2011 Paral. Processing*. Springer, 2011, pp. 3–14 (cit. on pp. 70, 72, 77).

[KR08] H. Koziolek and R. Reussner. “A model transformation from the palladio component model to layered queueing networks”. In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, pp. 58–78 (cit. on p. 37).

[KSS+17] J. Kienberger, S. Schmidhuber, C. Saad, S. Kuntz, and B. Bauer. “Parallelizing highly complex engine management systems”. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017), e4115 (cit. on p. 4).

[KTJ06] R. Kumar, D. M. Tullsen, and N. P. Jouppi. “Core architecture optimization for heterogeneous chip multiprocessors”. In: *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2006, pp. 23–32 (cit. on p. 22).

[Lar18] P. Laroche. *51st Known Mersenne Prime Discovered*. `https://www.mersenne.org/primes/press/M82589933.html`. (Accessed on 03/18/2020). Dec. 2018 (cit. on p. 88).

[LB14] S. Lehrig and M. Becker. “Approaching the Cloud: Using Palladio for Scalability, Elasticity, and Efficiency Analyses.” In: *SoSP*. 2014, pp. 141–151 (cit. on p. 33).

[LCFH14] I. Llopard, A. Cohen, C. Fabre, and N. Hili. “A Paral. Action Language for Embedded Applications and its Compilation Flow”. In: *Proceedings of the 17th Int. Workshop on Softw. and Compilers for Embedded Systems*. ACM. 2014, pp. 118–127 (cit. on pp. 71, 75).

[LCM+05] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[LE15] S. Lehrig and H. Eikerling. “Analyzing Cost-Efficiency of Cloud Computing Applications with SimuLizar”. In: *Proceedings of the Symposium on Software Performance 2015*. 2015 (cit. on p. 33).

[Leh18] S. M. Lehrig. *Efficiently conducting quality-of-service analyses by templating architectural knowledge*. Vol. 25. KIT Scientific Publishing, 2018 (cit. on pp. 7, 35, 36, 42–44, 117, 121, 150).

[LHB17] S. Lehrig, M. Hilbrich, and S. Becker. “The Architectural Template Method: Templating Architectural Knowledge to Efficiently Conduct Quality-of-Service Analyses”. In: spe.2517. 2017 (cit. on pp. 116, 122).

[LHB18] S. Lehrig, M. Hilbrich, and S. Becker. "The architectural template method: templating architectural knowledge to efficiently conduct quality-of-service analyses". In: *Software: Practice and Experience* 48.2 (2018), pp. 268–299 (cit. on pp. 7, 43).

[Lin10a] J. Link. "Paradigmen für Nebenläufigkeit - Teil 1: Transaktionaler Speicher und Unveränderlichkeit". In: *JavaSpektrum.* (Accessed on 03/23/2020). Apr. 2010 (cit. on p. 92).

[Lin10b] J. Link. "Paradigmen für Nebenläufigkeit - Teil 2: Aktoren, Parallele Collections und Data Flow". In: *JavaSpektrum.* (Accessed on 03/23/2020). Apr. 2010 (cit. on p. 93).

[LL12] J. Ludewig and H. Lichter. *Softw. Engi.: Grundlagen, Menschen, Prozesse, Techniken (German Edition).* dpunkt.verlag, 2012.

[LLL+11] C.-S. Lin, C.-H. Lu, S.-W. Lin, Y.-R. Chen, and P.-A. Hsiung. "VERTAF/Multi-Core: A SysML-based Application Framework for Multi-Core Embedded Softw. Development". In: *Journal of Computer Science and Technology* 3 (2011), pp. 448–462 (cit. on pp. 71, 75).

[Lue08] D. Luebke. "CUDA: Scalable Paral. Programming for High-Perf. Sientific Comp." In: *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE Int. Sym. on.* IEEE. 2008, pp. 836–838 (cit. on pp. 71, 74, 77).

[MBB+12] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, et al. "SPEC OMP2012—an application benchmark suite for parallel systems using OpenMP". In: *International Workshop on OpenMP.* Springer. 2012, pp. 223–236 (cit. on p. 95).

[McK11] P. E. McKenney. "Is Paral. Programming Hard, And, If So, What Can You Do About It?" In: *Linux Technology Center, IBM Beaverton* (2011).

[Mey11] J. Meyer. "Modellgetriebene Skalierbarkeitsanalyse von selbstadaptiven komponentenbasierten Softwaresystemen in der Cloud". In: *University of Paderborn, Masterarbeit* (2011), pp. 1–131 (cit. on p. 39).

[MGF11] G. Martinez, M. Gardner, and W.-c. Feng. “CU2CL: A CUDA-to-OpenCL Translator for Multi-and Many-Core Architectures”. In: *Paral. and Distr. Systems (ICPADS), 2011 IEEE 17th Int. Conf. on.* IEEE. 2011, pp. 300–307 (cit. on pp. 71, 74, 77).

[Mic17] Microsoft. *Pipes and Filters pattern - Cloud Design Patterns | Microsoft Docs.* `https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters`. (Accessed on 02/14/2020). June 2017 (cit. on p. 28).

[Mic19] S. Microsystems. *Java 8 Streams.* Mar. 2019. URL: `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html` (cit. on p. 125).

[MJU+09] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. “Multicore Compilation Strategies and Challenges”. In: *Signal Processing Magazine, IEEE* 6 (2009), pp. 55–63 (cit. on pp. 70, 72, 76).

[MMG+09] P. E. McKenney, M. M. Michael, M. Gupta, P. W. Howard, J. Triplett, and J. Walpole. “Is parallel programming hard, and if so, why?” In: (2009) (cit. on pp. 13, 24).

[MRR12] M. D. McCool, J. Reinders, and A. Robison. *Structured parallel programming patterns for efficient computation.* Elsevier/Morgan Kaufmann, 2012 (cit. on p. 126).

[MSB+05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. “Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset”. In: *SIGARCH Comput. Arch. News* 33.4 (Nov. 2005), pp. 92–99.

[MSM04] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming.* Pearson Education, 2004 (cit. on pp. 12, 19–23, 25, 27, 70, 72, 76, 123, 125, 126, 135).

[Mun09] A. Munshi. “The OpenCL Specification”. In: 2. Khronos OpenCL Working Group, 2009.

[MVL+10] M. S. Müller, M. Van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, et al. "SPEC MPI2007—an application benchmark suite for parallel systems using MPI". In: *Concurrency and Computation: Practice and Experience* 22.2 (2010), pp. 191–205 (cit. on pp. 94, 95).

[ÖGW+14] P.-O. Östberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth, et al. "The CACTOS vision of context-aware cloud topology optimization and simulation". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE. 2014, pp. 26–31 (cit. on p. 33).

[One09] M. e. O'neill. "The genuine sieve of Eratosthenes". In: *Journal of Functional Programming* 19.1 (2009), pp. 95–106 (cit. on p. 88).

[OPT09] F. Otto, V. Pankratius, and W. F. Tichy. "High-level Multicore Programming with XJava". In: *Softw. Engi.-Companion V., 2009. ICSE-Companion 2009. 31st Int. Conf. on*. IEEE. 2009, pp. 319–322 (cit. on pp. 70, 73, 76).

[Pac97] P. S. Pacheco. *Paral. Programming with MPI*. Morgan Kaufmann, 1997.

[PAG11a] A. Patel, F. Afram, and K. Ghose. "MARSS-x86: A QEMU-Based Micro-Architectural and Systems Simulator for x86 Multicore Processors". In: *Proc. 2011 1st Int. QEMU Users' Forum* 1 (2011), pp. 29–30.

[PAG11b] A. Patel, F. Afram, and K. Ghose. "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors". In: *1st International Qemu Users' Forum*. 2011, pp. 29–30 (cit. on p. 236).

[Pan11] V. Pankratius. "Automated Usability Evaluation of Paral. Programming Constructs (NIER track)". In: *Proceedings of the 33rd Int. Conf. on Softw. Engi.* ACM. 2011, pp. 936–939 (cit. on pp. 70, 73, 76).

[PBKW85] G. Pawley, K. Bowler, R. Kenway, and D. Wallace. "Concurrency and parallelism in MC and MD simulations in physics". In: *Computer Physics Communications* 37.1-3 (1985), pp. 251–260.

[PF02] S. Pllana and T. Fahringer. "UML based modeling of performance oriented parallel and distributed applications". In: *Proceedings of the Winter Simulation Conference*. Vol. 1. IEEE. 2002, pp. 497–505 (cit. on p. 80).

[PF05] S. Pllana and T. Fahringer. "Performance prophet: A performance modeling and prediction tool for parallel and distributed programs". In: *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE. 2005, pp. 509–516 (cit. on pp. 71, 75, 77, 78, 80).

[PH11] V. Pankratius and M. Heneka. "Moving Database Systems to Multicore: An Auto-Tuning Approach". In: *Paral. Processing (ICPP), 2011 Int. Conf. on.* IEEE. 2011, pp. 582–591 (cit. on pp. 70, 72, 77).

[Pie16] F. Pierfederici. *Distributed Computing with Python.* 2016 (cit. on p. 19).

[PJT09] V. Pankratius, A. Jannesari, and W. F. Tichy. "Paral.izing bzip2: A case study in multicore Softw. Engi." In: *Softw., IEEE* 6 (2009), pp. 70–77 (cit. on pp. 70, 72, 76).

[PPBK11] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan. "A Eeview on Task Perf. Prediction in Multi-Core Based Systems". In: *Computer and Information Technology (CIT), 2011 IEEE 11th Int. Conf. on.* IEEE. 2011, pp. 615–620.

[PRE00] I. PRESENT. "Cramming More Components onto Integrated Circuits". In: *Readings in computer architecture* (2000).

[PSJT08] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. "Softw. Engi. for Multicore Systems: An Experience Report". In: *Proceedings of the 1st Int. workshop on Multicore Softw. Engi.* ACM. 2008, pp. 53–60 (cit. on pp. 70, 72, 76).

[RBH+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and simulating software architectures: The Palladio approach.* MIT Press, 2016 (cit. on pp. 33, 35, 197).

[Rei07] J. Reinders. *Understanding task and data parallelism | ZDNet.* https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/. (Accessed on 02/03/2020). Sept. 2007 (cit. on p. 14).

[RFS16] P. Richter, M. Frank, and H. Schlieter. “Entwicklung eines Leitlinienmanagementsystems - Anforderungen und konzeptuelle Vorarbeiten”. In: *Multikonferenz Wirtschaftsinformatik (MKWI) 2016 - Technische Universitaet Ilmenau, 09. - 11. Maerz 2016; Band II.* Mar. 2016, pp. 679–690.

[RGD11a] A. W. D. O. Rodrigues, F. Guyomarc’h, and J.-L. Dekeyser. “A Modeling Approach Based on UML/MARTE for GPU Architecture”. In: *arXiv preprint arXiv* (2011) (cit. on pp. 70, 73).

[RGD11b] W. Rodrigues, F. Guyomarc’h, and J.-L. Dekeyser. “Programming Massively Paral. Architectures using MARTE: a Case Study”. In: *arXiv preprint arXiv:1103.4881* (2011) (cit. on pp. 7, 70, 73, 77, 79).

[RH09] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical software engineering* 14.2 (2009), p. 131 (cit. on p. 139).

[RHJ09] R. Rabenseifner, G. Hager, and G. Jost. “Hybrid MPI/OpenMP Paral. Programming on Clusters of Multi-Core SMP Nodes”. In: *Paral., Distr. and Net.-based Processing, 2009 17th Euromicro Int. Conf. on.* IEEE. 2009, pp. 427–436 (cit. on pp. 71, 74, 77).

[RQZ07] C. Rupp, S. Queins, and B. Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 2007 (cit. on p. 34).

[RR07] T. Rauber and G. Rünger. *Multicore:: Parallele Programmierung*. Springer-Verlag, 2007 (cit. on pp. 16, 24).

[RR12] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer-Verlag, 2012 (cit. on p. 16).

[SAB18] T. Sterling, M. Anderson, and M. Brodowicz. “Chapter 8 - The Essential MPI”. In: *High Performance Computing*. Ed. by T. Sterling, M. Anderson, and M. Brodowicz. Boston: Morgan Kaufmann, 2018, pp. 249–284. URL: http://www.sciencedirect.com/science/article/pii/B9780124201583000083 (cit. on p. 17).

[Sch08] B. Schauer. “Multicore processors–a necessity”. In: *ProQuest discovery guides* (2008), pp. 1–14 (cit. on pp. 24, 198).

[SEE19] A. Saad, A. El-Mahdy, and H. El-Shishiny. “Performance Modeling of MPI-based Applications on Cloud Multicore Servers”. In: *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*. 2019, pp. 1–6 (cit. on pp. 71, 74, 77, 80).

[SEPA13] G. Sagardui, L. Etxeberria, T. Perez, and J. A. Agirre. “Early Estimation of Quality Attributes in Multi-Core Systems”. In: *Actas de las IV Jornadas de Computación Empotrada (JCE)* (2013), p. 20 (cit. on pp. 70, 73).

[SK13a] D. Sanchez and C. Kozyrakis. “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems”. In: *Proc. Int. Symp. Comput. Archit.* (2013), pp. 475–486 (cit. on p. 30).

[SK13b] D. Sanchez and C. Kozyrakis. “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems”. In: *ACM SIGARCH Computer architecture news* 41.3 (2013), pp. 475–486 (cit. on p. 234).

[SKK+15] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. “Tejas: A java based versatile micro-architectural simulator”. In: *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2015, pp. 47–54 (cit. on p. 232).

[Smi80] C. U. Smith. “The prediction and evaluation of the performance of software from extended design specifications”. PhD thesis. University of Texas at Austin, 1980.

[Söh18] S. Söhnel. “Achieving the perfect Speedup - Why it is not possible”. Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2018 (cit. on pp. 158, 271).

[Som10] I. Sommerville. *Softw. Engi. (9th Edition)*. Pearson, 2010.

[SPE20] B. SPEC. *SPEC Benchmark Website*. 2020. URL: `http://www.spec.org/index.html` (cit. on p. 94).

[SPT10] C. A. Schaefer, V. Pankratius, and W. F. Tichy. “Engi. Paral. Applications with Tunable Architectures”. In: *Proceedings of the 32nd ACM/IEEE Int. Conf. on Softw. Engi.-V. 1.* ACM. 2010, pp. 405–414 (cit. on pp. 70, 72, 77).

[SSA18] M. Stach, D. Steinacker, and K. Allhyarli. “Simulators for Cloud Environments”. Seminar Theses at the Reliable Software Systems Group at University Stuttgart. Aug. 2018.

[Sta17] S. Staude. “Evaluation des Palladio Component Model (PCM) bzgl. der Eignung für Mehrkernprozessorsysteme anhand der Fallbeispiele Matrixmultiplikation und Bankmanagement”. Bachelor Theses at the Chair of Software Engineering at the Technische Unversität Chemnitz. May 2017 (cit. on p. 271).

[Sta73] H. Stachowiak. *Allgemeine modelltheorie.* Springer, 1973 (cit. on pp. 109, 197).

[Sto15] B. Storti. *The actor model in 10 minutes.* `https://www.brianstorti.com/the-actor-model/`. (Accessed on 02/10/2020). July 2015 (cit. on p. 18).

[Sun08] I. Sun Microsystems. *Multithreaded Programming Guide.* Sun Microsystems, Inc., 2008 (cit. on p. 12).

[SV12] C. Sonnenberg and J. Vom Brocke. “Evaluations in the science of the artificial–reconsidering the build-evaluate pattern in design science research”. In: *International Conference on Design Science Research in Information Systems.* Springer. 2012, pp. 381–397 (cit. on p. 58).

[SW01] C. U. Smith and L. G. Williams. “Perf. Solutions: A Practical Guide to Creating Responsive, Scalable Softw.” In: (2001).

[SW03] C. U. Smith and L. G. Williams. “Software performance engineering”. In: *UML for Real.* Springer, 2003, pp. 343–365 (cit. on p. 29).

[SW17] R. Sedgewick and K. Wayne. *Mandelbrot.java.* `https://introcs.cs.princeton.edu/java/32class/Mandelbrot.java.html`. (Accessed on 03/18/2020). Oct. 2017 (cit. on p. 86).

[SWD19] J. Schuder, A. Weller, and Z. Denis. “Identifying Reoccurring Parallel Design Patterns to Build a Parallel Pattern Catalogue for Palladio”. Seminar Theses at the Reliable Software Systems Group at University Stuttgart. Aug. 2019 (cit. on pp. 25, 123, 124, 128, 129, 271).

[SWH+09] M. Stürmer, G. Wellein, G. Hager, H. Köstler, and U. Rüde. “Challenges and Potentials of Emerging Multicore Architectures”. In: *HPC in Science and Engi., Garching/Munich 2007*. Springer, 2009, pp. 551–566 (cit. on pp. 70, 73).

[Tec17] TechDifferences. *Difference Between Concurrency and Parallelism (with Comparison Chart) - Tech Differences*. `https://techdifferences.com/difference-between-concurrency-and-parallelism.html`. (Accessed on 01/02/2020). Dec. 2017 (cit. on p. 13).

[THW09] J. Treibig, G. Hager, and G. Wellein. “Multi-Core Architectures: Complexities of Perf. Prediction and the Impact of Cache Topology”. In: *arXiv preprint arXiv:0910.4865* (2009) (cit. on pp. 7, 71, 76, 77, 80).

[THW12] J. Treibig, G. Hager, and G. Wellein. “Perf. Patterns and Hardware Metrics on Modern Multicore Processors: Best Practices for Perf. Engi.” In: *European Conf. on Paral. Processing*. Springer. 2012, pp. 451–460 (cit. on pp. 7, 71, 76).

[TMCB16] C. Terboven, M. S. Müller, B. Chapman, and C. Bischof. *Abstractions for Performance Programming on Multi-Core Architectures with Hierarchical Memory*. RWTH-2016-06261. Fachgruppe Informatik, 2016 (cit. on pp. 70, 73).

[Tru20] K. Truong. “Extending the Palladio Meta-Model to Support Memory Hierarchy”. Master theses Theses at the Reliable Software Systems Group at University Stuttgart. 2020 (cit. on pp. 194, 196, 207, 271, 307–310).

[UJM+12] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. “Multi2Sim: a simulation framework for CPU-GPU computing”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2012, pp. 335–344 (cit. on p. 236).

[VE11] K. Van Craeynest and L. Eeckhout. “The Multi-program Perf. Model: Debunking Current Practice in Multi-Core Simulation”. In: *Workload Characterization (IISWC), 2011 IEEE Int. Sym. on.* IEEE. 2011, pp. 26–37 (cit. on pp. 7, 71, 76, 77, 80).

[Ver15] V. Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka.* Addison-Wesley Professional, 2015 (cit. on p. 18).

[VFB20] V. Vijayshree, M. Frank, and S. Becker. “Extended Abstract of Performance Analysis and Prediction of Model Transformation”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 8–9.

[WBKK15] F. Willnecker, A. Brunnert, B. Koch-Kemper, and H. Krcmar. “Full-stack performance model evaluation using probabilistic garbage collection simulation”. In: *Proceedings of the 2015 Symposium on Software Performance (SSP 2015).* 2015 (cit. on p. 205).

[WHW12] A. Wert, J. Happe, and D. Westermann. “Integrating software performance curves with the palladio component model”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. 2012, pp. 283–286 (cit. on p. 158).

[Wil09] S. Williams. “Roofline: An Insightful Visual Perf. Model for Floating-Point Programs and Multicore”. In: (2009) (cit. on pp. 7, 71, 76, 77, 80).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 139, 146).

[WS03] L. G. Williams and C. U. Smith. “Making the Business Case for Software Performance Engineering”. In: *Int. CMG Conference.* 2003, pp. 349–358 (cit. on pp. 5, 33).

[XCDM10] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. “Cache Contention and Application Perf. Prediction for Multi-Core Systems”. In: *Perf. Analysis of Systems & Softw. (ISPASS), 2010 IEEE Int. Sym. on.* IEEE. 2010, pp. 76–86 (cit. on pp. 7, 71, 76, 77, 80).

[Yar09] V. Yaroslavskiy. “Dual-pivot quicksort algorithm”. In: *Retrieved December* 24 (2009), p. 2018 (cit. on p. 86).

[Yoo19] J.-A. Yoon. "Performance Benchmarks for Parallel Business Information Systems (BIS)". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2019 (cit. on pp. 93, 271).

[Zah20] D. Zahraive. "Controlled User Study: Usability and Efficiency Evaluation of the Parallel Performance Catalogue Extension for the Palladio-Bench". B.S. thesis. 2020 (cit. on pp. 139, 145, 147, 271).

[ZP12] A. Zwinkau and V. Pankratius. "AutoTunium: An Evolutionary Tuner for General-Purpose Multicore Applications". In: *Paral. and Distr. Systems (ICPADS), 2012 IEEE 18th Int. Conf. on*. IEEE. 2012, pp. 392–399 (cit. on pp. 70, 72, 77).

All URLs were last checked on September 28, 2021

Publications of the Author

[FBKK19] M. Frank, S. Becker, A. Kaplan, and 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*. ACM. 2019, pp. 21–24 (cit. on pp. 84, 154, 158–160, 162, 195, 271).

[FH16] M. Frank and M. Hilbrich. "Performance Prediction for Multicore Environments—An Experiment Report". In: *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany*. 2016. URL: `https://sdqweb.ipd.kit.edu/typo3/sdq/fileadmin/user_upload/palladio-conference/2016/papers/Performance_Prediction_for_Multicore_Environments_-_An_Experiment_Report.pdf` (cit. on pp. 7, 49, 52, 59, 61, 62, 81, 91, 100–103, 135, 151, 195, 211–213, 216, 249, 271, 325).

[FH18] M. Frank and A. Hakamian. "An Architectural Template for Parallel Loops and Sections". In: *Proceedings of the Symposium on Software Performance 2018, 7-9 November 2018, Hildesheim, Germany*. 9th Symposium on Software Performance 2018. Hildesheim, Nov. 2018. URL: `https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2018/papers/FrankHakamian18.pdf` (cit. on pp. 130, 271).

[FHB20] M. Frank, A. Hakamian, and S. Becker. "Defining a Formal Semantic for Parallel Patterns in the Palladio Component Model Using Hierarchical Queuing Petri Nets". In: *Software Architecture*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, and U. Zdun. Cham: Springer International Publishing, 2020, pp. 381–394 (cit. on p. 100).

[FHL15] M. Frank, M. Hilbrich, and S. Lehrig. “Improved Scalability for Job-centric Monitoring in Distributed Infrastructures”. In: *CGW Workshop 15 Proceedings*. ISBN 978-83-61433-14-9. ACC Cyfronet AGH, Oct. 2015, pp. 79–80.

[FHLB17] M. Frank, M. Hilbrich, S. Lehrig, and 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 pp. 7, 61, 64, 65, 67, 71, 73–76, 78, 271).

[FKB18] M. Frank, F. Klinaku, and 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 (cit. on pp. 63, 76, 271).

[FKHB19] M. Frank, F. Klinaku, M. Hilbrich, and 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 pp. 100, 106, 107, 271).

[FSH17] M. Frank, S. Staude, and M. Hilbrich. “Is the PCM Ready for ACTORs and Multicore CPUs? - A Use Case-based Evaluation”. In: *Proceedings of the Symposium on Software Performance 2017, 9-10 November 2017, Karlsruhe, Germany*. 8th Symposium on Software Performance 2017. Karlsruhe, Nov. 2017. URL: `http://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2017/papers/Is_the_PCM_Ready_for_ACTORs_and_Multicore_CPUs_A_Use_Case-based_Evaluation.pdf` (cit. on pp. 7, 49, 52, 55, 59, 61, 62, 151, 245, 246, 249, 271, 325).

[FSK+20] M. Frank, L. Schmid, A. Kaplan, L. Greiner, A. Koziolek, and S. Becker. “Performance Curves for better Performance Predictions of Parallel Applications in Multicore Environments”. In: *Companion of the 2020 ACM/SPEC International Conference on Performance Engineering*. currently under review. ACM. 2020 (cit. on p. 154).

[GF19] P. Gruber and M. Frank. “Modelling and Predicting Memory Behaviour in Parallel Systems with Network Links—Palladio-based Experiment Report”. In: *Proceedings of the Symposium on Software Performance 2019, 4-6 November 2019, Wuerzburg, Germany*. 10th Symposium on Software Performance 2019. Wuerzburg, Nov. 2019. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2019/Papers/SSP2019_paper_3.pdf (cit. on pp. 194–196, 271).

[HF15a] M. Hilbrich and M. Frank. “Analysis of Series of Measurements from Job-Centric Monitoring by Statistical Functions”. In: *CGW Workshop 15 Proceedings*. ISBN 978-83-61433-14-9. ACC Cyfronet AGH, Oct. 2015, pp. 81–82.

[HF15b] M. Hilbrich and M. Frank. *Better Resource Usage by Job-Centric Monitoring*. Symposium on Software Performance 2015, Poster, Munich, Germany. Nov. 2015.

[HF17a] M. Hilbrich and M. Frank. “Analysis of Series of Measurements from Job-Centric Monitoring by Statistical Functions”. In: *Computer Science* 18.1 (2017), p. 2. URL: https://journals.agh.edu.pl/csci/article/view/1791.

[HF17b] M. Hilbrich and M. Frank. “Debugging a Complex Systems, the Long Way from Data to Knowledge”. In: *Proceedings of the Symposium on Software Performance 2017, 9-10 November 2017, Karlsruhe, Germany*. 8th Symposium on Software Performance 2017. Karlsruhe, Nov. 2017. URL: http://www.performance-symposium.org/2017/program/debugging-a-complex-systems-the-long-way-from-data-to-knowledge/.

[HF17c] M. Hilbrich and M. Frank. “Enforcing Security and Privacy via a Cooperation of Security Experts and Software Engineers-a Model-based Vision”. In: *Proceedings of the 7th IEEE International Symposium on Cloud and Service Computing, 22-25 November 2017, Kanazawa, Japan*. 7th IEEE International Symposium on Cloud and Service Computing. Kanazawa, Nov. 2017.

[HF18a] M. Hilbrich and M. Frank. “Abstract Fog in the Bottle-Trends of Computing in History and Future”. In: *2018 44th Euromicro*

Conference on Software Engineering and Advanced Applications (SEAA). IEEE. 2018, pp. 519–522.

[HF18b] M. Hilbrich and M. Frank. “HPC and SPE Need to Learn from Each Other-Knowledgetransformation Patterns”. In: *The 18th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations*. SCAN2018 Book of Abstracts. Tokyo, Japan, 2018, pp. 58–59.

[HFL16] M. Hilbrich, M. Frank, and S. Lehrig. “Security Modeling with Palladio—Different Approaches”. In: *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany*. 2016. URL: https://sdqweb.ipd.kit.edu/typo3/sdq/fileadmin/user_upload/palladio-conference/2016/papers/Security_Modeling_with_Palladio-Different_Approaches.pdf (cit. on p. 33).

[HLF16] M. Hilbrich, S. Lehrig, and M. Frank. *Measured Values Lost in Time—or How I rose from a User to a Developer of Palladio*. http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-213813. 2016.

[KFB18] F. Klinaku, M. Frank, and S. Becker. “CAUS: An Elasticity Controller for a Containerized Microservice”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 93–98.

[KHF19] F. Klinaku, A. Hakamian, and M. Frank. “A Process Model for Elastic and Resilient IoT Applications with Emergent Behaviors”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pp. 27–30.

[RFS16] P. Richter, M. Frank, and H. Schlieter. “Entwicklung eines Leitlinienmanagementsystems - Anforderungen und konzeptuelle Vorarbeiten”. In: *Multikonferenz Wirtschaftsinformatik (MKWI) 2016 - Technische Universitaet Ilmenau, 09. - 11. Maerz 2016; Band II*. Mar. 2016, pp. 679–690.

[VFB20] V. Vijayshree, M. Frank, and S. Becker. “Extended Abstract of Performance Analysis and Prediction of Model Transformation”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 8–9.

All URLs were last checked on September 28, 2021

Supervised Theses

[Ant19] I. Antunovic. "Simulation, Modeling and Verification Environment for Multicore Software Architectures for Automotive Embedded Systems". Master theses Theses at the Reliable Software Systems Group at University Stuttgart. May 2019.

[Det20] S. Dettenmaier. "Integrating and Automating a Multicore CPU Simulator into Palladio". B.S. thesis. 2020 (cit. on pp. 228, 271).

[Gra18] S. Graef. *Connecting Palladio with multicore CPU simulators*. Englisch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Sichere und Zuverlässige Softwaresysteme. Bachelorarbeit. Oct. 2018. URL: `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-2018-70&engl=0` (cit. on pp. 30, 228, 231, 232, 234–237, 240, 241, 243, 271, 318, 319, 321).

[Gre19] L. Greiner. "Bewertung verschiedener Parallelisierungsstrategien im Hinblick auf Leistungsfähigkeit von paralleler Programmausführung". Bachelor Theses at the Chair for Software Design and Quality in Karlsruhe in collaboration with the Reliable Software Systems Group at University of Stuttgart. 2019 (cit. on pp. 158, 165, 167, 169, 170, 271, 286–303).

[Gru19] P. Gruber. "Using Palladio network links to model multicore architecture memory hierarchies". B.S. thesis. 2019 (cit. on pp. 194, 200, 212, 271).

[Gru20] P. Gruber. "Benchmarking and Measuring of Performance-influencing Factors". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2020 (cit. on pp. 93, 271).

[Söh18] S. Söhnel. "Achieving the perfect Speedup - Why it is not possible". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2018 (cit. on pp. 158, 271).

[SSA18] M. Stach, D. Steinacker, and K. Allhyarli. "Simulators for Cloud Environments". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. Aug. 2018.

[Sta17] S. Staude. "Evaluation des Palladio Component Model (PCM) bzgl. der Eignung für Mehrkernprozessorsysteme anhand der Fallbeispiele Matrixmultiplikation und Bankmanagement". Bachelor Theses at the Chair of Software Engineering at the Technische Unversität Chemnitz. May 2017 (cit. on p. 271).

[SWD19] J. Schuder, A. Weller, and Z. Denis. "Identifying Reoccurring Parallel Design Patterns to Build a Parallel Pattern Catalogue for Palladio". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. Aug. 2019 (cit. on pp. 25, 123, 124, 128, 129, 271).

[Tru20] K. Truong. "Extending the Palladio Meta-Model to Support Memory Hierarchy". Master theses Theses at the Reliable Software Systems Group at University Stuttgart. 2020 (cit. on pp. 194, 196, 207, 271, 307–310).

[Yoo19] J.-A. Yoon. "Performance Benchmarks for Parallel Business Information Systems (BIS)". Seminar Theses at the Reliable Software Systems Group at University Stuttgart. 2019 (cit. on pp. 93, 271).

[Zah20] D. Zahraive. "Controlled User Study: Usability and Efficiency Evaluation of the Parallel Performance Catalogue Extension for the Palladio-Bench". B.S. thesis. 2020 (cit. on pp. 139, 145, 147, 271).