Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Using Palladio Network Links to Model Multicore Architecture Memory Hierarchies

Philipp Gruber

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Markus Frank, M.Sc.
Commenced:	Oktober 30, 2018
Completed:	April 30, 2019

Abstract

This thesis investigates the capabilities of Palladio to predict the performance of software/hardware systems. The Palladio simulations are accurate for systems which run on single core processors. Experiments showed that the predictions are not accurate for multicore systems. The parallelization of programs is complex. In addition a parallelized program executed on four cores is not automatically four times faster than the single core program. There are reasons for this on the software/code side (e.g. Amdahl´s law) but also on the hardware side (e.g. memory bandwidth). The so called memory bandwidth is referring to the capacity limit of the memory bus, the bus from the CPU to the memory. The memory bandwidth is theoretically becoming a more important factor by an increasing degree of parallelization. More cores lead to the fact that more data is flowing in shorter time that bus. A consequence is that memory bandwidth becomes a bottleneck because of an over strained memory bus, which leads to idle CPU´s. Due to the fact that they have to wait to load data from the memory. Such effects of multicore systems are not taken into account by Palladio.

This thesis had the target to find out, if Palladio is able to model the memory bandwidth with existing elements from their component model and subsequently if this modeling leads to more accurate predictions. Our work showed on the basis of an experiment with a matrix multiplication that this is possible, but we are not able to reach the 100% accuracy with our approach. The achieved accuracy of approximately 90% in average indicates the existence of more factors which contribute to the non-linear speedup of multicore processors. Examples are the synchronization of shared memory or the contention for these resources. In addition our experiments lacked in confidence to determine in which quantity the memory bandwidth was a bottleneck for our specific use-case. This should be the target of future work.

Kurzfassung

Diese Arbeit erforscht die Fähigkeiten von Palladio um die Leistungsfähigkeit von Software/Hardware Systemen vorherzusagen. Die Palladio Vorhersagen/Simulationen haben sich im Fall von Einkernprozessoren als akkurat erwiesen, allerdings haben Experimente gezeigt, dass dies nicht für Mehrkernprozessoren gilt. Die Parallelisierung von Programmen ist komplex. Zusätzlich ist ein parallelisiertes Programm, welches auf vier Kernen ausgeführt wird, nicht automatisch viermal schneller als eines auf einem Kern. Dies hat nicht nur Ursachen auf Code/Softwareebene, sondern wird auch durch Hardware Faktoren, wie unter anderem der Speicherbandbreite ausgelöst. Mit Speicherbandbreite ist die Kapazitätsgrenze des Bus vom CPU Kern bis zum Speicher gemeint, welche zum Flaschenhals wird. Dies bedeutet der Rechenkern muss warten bis er Werte laden oder speichern kann. Dieser Einflussfaktor nimmt theoretisch zu, je mehr Daten in kürzerer Zeit über den Bus fließen, sprich bei einem höheren Grad an Parallelisierung. Dies wird bei der bisherigen Realisierung von Mehrkernsystemen in Palladio nicht berücksichtigt.

Diese Arbeit hatte das Ziel herauszufinden, ob Palladio die Speicherbandbreite mit bestehenden Mitteln/Elementen abbilden kann und ob dies zu einer höheren Genauigkeit führt. Das Ergebnis anhand eines Experiments mit einer Matrix Multiplikation hat gezeigt, dass es sowohl möglich ist wie auch zu einer größeren Genauigkeit führt, wenn dieser Ansatz verwendet wird. Dennoch gelang es bei unserem Experiment nicht eine Genauigkeit von annähernd 100% zu erzielen, sondern im Durchschnitt nur circa 90%. Dies deutet darauf hin, dass neben der Speicherbandbreite noch andere Faktoren eine Rolle für die nicht lineare Beschleunigung der Programme spielen. Zum Beispiel die Synchronisation der geteilten Speicher oder der Wettbewerb um die geteilten Ressourcen. Ebenso fehlt die Gewissheit, dass und in welcher Form der Speicherbus in unserem Anwendungsfall tatsächlich vollkommen ausgelastet war und zum Flaschenhals wurde. Dies müsste in der weiteren Forschung berücksichtigt und genauer untersucht werden.

Contents

List of Figures

Chapter 1

Introduction

In the design stages of complex, connected and large software systems, software architects are forced to take certain design decisions. Without seeing their immediate effect, because quality aspects of software systems as i.e. performance and reliability get visible after the implementation [RBH+16]. Thus, these decisions have a direct impact on the quality of the software. If poorly designed architectures are implemented, expensive redesigns are the consequence [RBK+07]. Software quality prediction tools help to prevent respectively detect poor design decisions. They enable to estimate the effect of design decisions and evaluate alternatives against each other [FHLB17].

Palladio Bench is one state of the art software quality prediction tool. It uses a component model-based approach to evaluate quality attributes based on the architectural design and is therefore suited for the use in early design stages. One aspect of Palladio for example is the performance prediction. Palladio is able to predict performance metrics as response times, resource demand (temporal/spatial) and utilization of component based software architectures. With the help of CPU metrics, HDD access rates and network links of the respective components. Therefore it is even useful for complex systems [BKR09].

[BKR09] further pointed out there are several limitations to the PCM (Palladio Component Model), such as limited support for concurrency. It is possible to model multicore processor systems but their performance prediction calculation model assumes a linear speedup with an increasing number of cores.

1.1. Problem

The fact that PCM has limitations regarding predicting performance for multicore systems implies PCM is inaccurate at evaluating multicore processor systems. For single core

systems it is accurate enough using only CPU speed as a performance metric and to ignore memory effects. But single core system behave differently than multicore systems. In addition to the mentioned memory effects, especially caching and memory bandwidth are expected to make a difference in the performance prediction. Memory bandwidth is further claimed as a game changer by [FKB18] because they observed such effects. These effects are part of the fact that the speedup with an increasing number of cores is non-linear.

In case of the inaccuracy we are able to rely on work from other authors, who give us a foundation, see section 3.2. They did not solved the problem, but they made first experiments to examine the inaccuracy, tried solutions with other approaches and showed that with their solution PCM is not ready for multicore systems.

1.1.1. Idea

The problem of inaccurate predictions seems to be caused by memory bandwidth and memory "hierarchies" effects. The idea of this thesis is to use existing Palladio objects. Those elements should have similar attributes, which allows to remodel them. As [FKB18] pointed out "limited" memory bandwidth is simplified a capacity limited memory bus. Thus the memory bus/memory bandwidth has a similarity to a network link.

Palladio is able to model network links, therefore we aim to use the Palladio network links because of these similarities to a memory bus. In Palladio the specification of the network links regarding latency, throughput and failure probability is possible. Though, the adaption of these attributes for a memory bus is not trivial, because both elements are not equal. But due to the existence of this specification possibility it seems feasible to remodel the memory bandwidth effect. Given that the memory bus has a limited throughput, a certain latency (memory latency) and also a failure probability. This is the same with possible hardware architectures, e.g. trying to abstract from a multicore processor and then model its distinctive parts as "resource container" elements. Such as giving the cache/memory its own resource container. We also have to keep the other parts of the PCM in mind. Changes in the resource environment have an influence on the allocation, repository and system diagram. That depends on the element we changed. In addition, the composition of the components offer us the possibility to vary the architecture, if needed.

Apart from the memory bandwidth, the memory hierarchies as the second factor could be imitated by the passive resources from Palladio. Those are capable of acquiring and releasing a limited resource. In multicore systems not only the memory bus is shared. Caches (e.g. L3) are also shared, that leads to overhead effects in the synchronization or

at the resource contention in general. It seems feasible to model such a synchronization/contention effect in Palladio. Certainly [FKB18] stated that the memory design is complex and differs a lot from system to system, which is an issue. Furthermore from cache replace and update strategies, over garbage collection to hardware coherence protocols many factors are important for the execution time. This fast growing complexity lead to the decision to start with an simple architecture and to focus to model the memory bandwidth.

This description is a rough sketch of the idea of our approach.

1.1.2. Objectives

The general objective in this thesis is to evaluate Palladio's capabilities regarding representing modern multicore systems with the focus on the memory bandwidth. We have two main research questions and associated sub-items which we aim to investigate:

- RQ1) Is it possible to model multicore systems with the described approach?
  - 1.1) Is it possible to model memory bandwidth behavior of multicore systems with Palladio Bench by using network links?
  - 1.2) Is it possible to find representative for memory/cache and their size by using network links?
  - 1.3) Are there effects we did not consider and can we model them in Palladio?
  - 1.4) Are there missing parts/things in Palladio to successfully cope with multicore systems?
    - 1.1.1) How much work does it take to model our approach in Palladio?
    - 1.1.2) How many and which steps were needed to model our approach?
    - 1.1.3) Did problems occur and if how we solved them?
- RQ2) Is the prediction more accurate?
  - 2.1) How accurate are the results in comparison to the performance in reality?
  - 2.2) How accurate are the results in comparison to the "default" or single core Palladio simulations?
  - 2.3) How accurate are the results in comparison to the performance of other multicore simulation approaches?

From the research questions we formulated the following hypothesis:

- H1) We are able to model multicore systems with the network links.
- H2) The prediction is more accurate and reaches the 90 percent accuracy.

With the accuracy it is meant, that by using our idea, we are able to achieve at least 90% accuracy in comparison to the measured numbers by [FH16] or other experiments which measured the "response time" in reality.

1.2. Methodology

After introducing the problem, the idea and the objectives of this thesis, we step forward to explain on how we executed this study. Moreover we present the benefit, structure of our work and the process.

Both research questions are related to each other, in simple words the first covers the theoretical possibility and the second the accuracy gain of the possibility. We are only able to answer the second research question, if the answer to the first question is yes. The metric for the first question is complex because there are various factors, as we showed with the related sub-items. The quality of the approach is secondary, equal for usability and also the actual accuracy gain. The target is to model the memory bandwidth and cache (memory hierarchies) aspect with Palladio, as close as we can represent the reality in our model. In this case the question is: Are we able to model a bandwidth (byte/sec) in Palladio and does it change the prediction. This is the first level, then the "quality" and the second research question are worth to consider.

The metric for the accuracy gain (RQ2) is the response time of Palladio model in comparison to measurement of the corresponding use-case in reality. The difference of both values is represented in one percent value. In terms of how close the value of a solution is to the 100% accuracy, the better we judge its effectiveness.

1.2.1. Benefit

The benefit of this thesis is that we directly continue the research of [FH16] with the same experiment, but in a variation which has not been done yet. We target to deliver new knowledge, in order to examine the readiness of Palladio for multicore systems. In addition the solution with existing Palladio resources is instantly available and easier to realize than any extension, because meta model changes take long and are complex.

1.2.2. Process

The process itself and the method is realized in the following iteration steps (pictured in 1.1):

- **0. Iteration prototyping:** The first step was to model different prototypes regarding the possible resource container and network link structure mentioned above. At the end of this iteration we aimed to answer the question, which structures are promising and which not. The prototypes were evaluated against the old method to model multicore systems in palladio, by scaling up the replicas (cores) in the resource element. We tried also to model Memtest86 in Palladio as a baseline for the memory bandwidth in more complex use-cases. Baseline means that we were able to give an input of a certain number of bytes. The network has then a certain throughput "limit" which leads to a prediction how long it takes to send the input over the bus/network. With such prototypes we were able answer the first research question and then proceeded to the next iteration.
- **1. Iteration modeling:** The second step was to transfer the prototypes from iteration one to the experiment of [FH16]. We retried the Palladio modeling part of the experiment, but with combining the matrix multiplication with the Palladio modeling of the Memtest. On the one hand we calculate on the basis of matrix multiplication use-case, meaning "how much" memory bandwidth/caching effects should occur in theory for the different amount of cores. We plug these numbers into our model and calibrate the throughput on the basis of the Memtest86 results for the used hardware. After getting resilient and stable results we moved over to compare the simulation results (response time) with the fitting measurement and evaluate whether our approach brought an accuracy improvement to the results of [FH16] or not. This part includes discussions about the architecture with domain experts.
- **2. Iteration experiment:** After modeling the memory bandwidth/memory hierarchies we need a verification by repeating the experiment on another environment. This should give us more confidence regarding the validity. The verification includes a complete retry of [FH16]´s experiment. Moreover we have to do the whole process from the first iteration, conducting of hardware performance measurements, e.g. Memtest86. Then calibrate our approach with these numbers and simulate the modeling. At the end we compare these numbers with the measurements of the real execution. Thereby we can answer the second research question regarding potential accuracy gains.

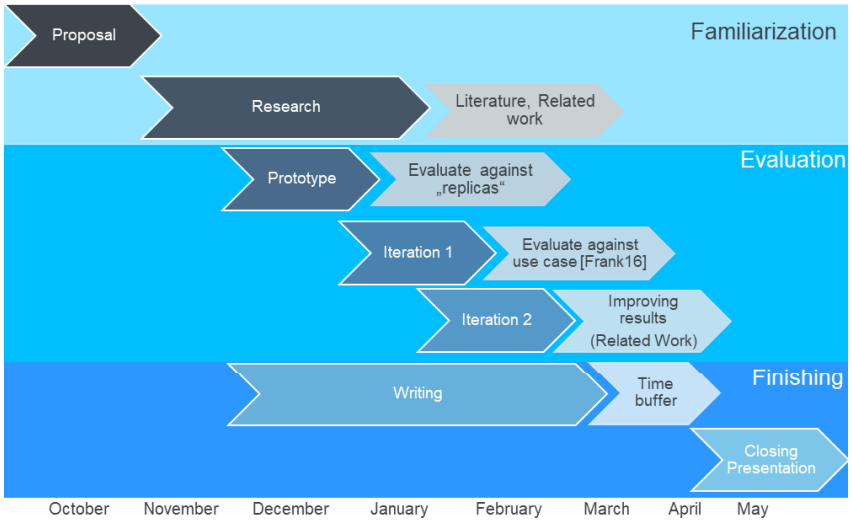

Figure 1.1.: Flow chart of the method

1.3. Outline

This thesis is structured in seven chapters and the appendix, they are characterized as follows:

Chapter 2 – Foundations: This chapters gives a compact overview of the principles regarding this thesis. The chapters offers introductions of multicore performance bottlenecks, Memtest86, Hardware Performance Counters and the Palladio Component Model (PCM).

Chapter 3 – Related work: The third chapter presents the related work as a detailed description of existing performance models and [FH16] experiment.

Chapter 4 – Approach to address shortcomings explains our approach to address the shortcomings of PCM´s performance predictions.

Chapter 5 – Experiment reports in detail about the conduction, the process and each of the three phases of this thesis.

Chapter 6 – Evaluation shows the results of our work. They will be discussed in order to answer our research questions with corresponding metrics and also judge about our hypothesis. The chapter also lists threats to the validity.

Chapter 7 – Conclusion summarizes the work and discusses it, and closes with an outlook about possible future research.

Chapter A – Appendix presents the results in a manner of detailed data and calculations.

Chapter 2

Foundations

This chapter contains information about multicore systems, their bottlenecks especially the memory bandwidth, techniques to measure hardware events and Palladio. Everything is crucial to understand this thesis.

2.1. Multicore systems

Multicore systems are state of the art for current computer systems/devices, even smart phones run on it [FHLB17]. The background of this development is one of the most popular laws in computer sciences, Moore´s Law. The law describes the improvement rate of CPU in a certain period of time. More precisely the doubling of the transistors counts or the CPU speed every 18 months [MOO75].

Until roughly 2003 the law was fulfilled by single core processors and an increasing clock frequency. But then problems occurred. The most common two are the fact that the effort cooling and energy consumption become too high. So single core processors reached an upper bound and were not able anymore of fulfilling Moore´s Law [FHLB17]. Therefore, CPU manufactures introduced multicore CPU´s and because of this shift Moore´s Law is still valid. The change from single to multicore created new challenges. On the software side it does not mean an automatic gain of performance. A multicore processor with two cores is not automatically twice as fast as a single core with a comparable characteristic. Programmers respectively software developers had to program parallel to use the possible performance gain. Parallel programming is more "expensive" than serial programming. Frameworks, such as ACTOR or OpenMP, are capable to help programmers to parallelize their programs.

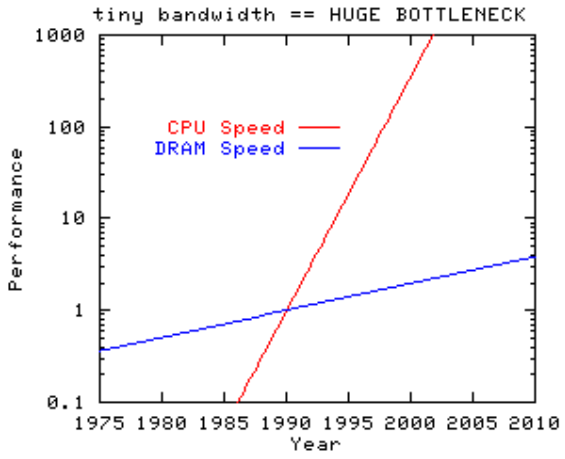

Figure 2.1.: Illustration of the grown gap between CPU and DRAM speed from [McC07].

2.1.1. Performance bottlenecks

However, the frameworks solve only a part of the problem, because multicore can only speed up programs which can work parallel. For example Amdahl´s law [HM08] addresses that fact. It describes the bottleneck of multicore programs on the software side, which is the amount of serial parts in those programs. But with improvements on the hardware side complex architectures can target the issues mentioned above. E.g. with a turbo boost mode for one core, which reaches higher clock frequency [HM17]. That enables CPU´s to speed up both serial and parallel parts, but they still remain the performance restricting part on software side.

In this work we deal mainly with performance issues on the hardware side: The limited bandwidth on the memory bus, which means roughly the ability to process what goes from the CPU into the memory [YP10], more precisely the data per time unit. The definitions of the term memory bandwidth are ambiguous. [DK00] explain the memory bandwidth with the help of two terms. The first term the machine balance is defined by them as: "The amount of data transfer the machine provides for each machine operation." Analog the definition of the program balance but now instead of provide, the program needs for every computation operation. The gap between program and machine balance is in this case the memory bandwidth and also the bottleneck. For our purpose this definition is sufficient.

Cache/memory layers as for example L2 cache partitioning helps to reduce the required memory bandwidth and the effects of it by the system [BXG12]. In general a current multicore CPU system is usually a multi core CPU with a complex architecture, different layer of cache/memory but still with limited bandwidth or MPI (Message passing interface). But as soon as the size of the object to load or write is higher than the cache

size, the system deals with the same tiny memory bandwidth of the main memory. The reason why this fact got more important over the years is the so called "memory wall" or processor (CPU) memory (DRAM) gap, see 2.1.

Even though caches reduced the influence. It describes that the speed of memory increased over the years much slower than the CPU speed. Therefore, caches on the processor are needed because a small/fast memory at or near the processor helps to reduce the impact of this speed gap. Regarding cache most of the processors have different level of caches, that is the memory hierarchy. The hierarchy starts at the L1 (Level 1) cache. The L1 cache is the private cache on the specific core. Normally, it depends on the processor, the L2 cache is in terms of size a bigger private cache but more far away and slower than the L1 cache. L3 caches then are shared between the cores. This structure brings up other challenges respectively potential bottlenecks which have to be solved, such as the cache coherence problem for shared cache. Or how to write through changes from private into shared memory and cache misses, which lead to the "time expensive" loading of data from a lower and slower memory hierarchy level. A "cold" cache leads to slower speed at the start of the use, too. The term "cold" refers to the fact, that at first the demanded data has to be load into the cache.

2.2. Memory benchmark programs

It is possible to calculate the maximal memory bandwidth of a computer system by the formula 2.1.

For the following processor [1] it is 25.6 GB/s, as shown in the hardware specification. For the corresponding calculation see A.1 in the appendix.

The theoretical maximum is unequal to the real memory bandwidth we face. This "real" or "effective" memory bandwidth has to be determined by measuring. In order to get a representative measure of the effective memory bandwidth there are several programs which enable this. For example the STREAM Benchmark or MemTest86.

Listing 2.1 Formula for max. Memory Bandwidth.

```
max. Memory Bandwidth =
Base DRAM clock frequency * Number of data transfer per clock *
Memory bus interface width * Number of interfaces
```

[1] https://ark.intel.com/content/www/us/en/ark/products/64896/intel-core-i5-3320m-processor-3m-cache-up-to-3-30-ghz.html

2.2.1. MemTest86

MemTest86 [2] is a memory testing software tool from the company PassMark software. Before PassMark took over it was a open source software under GPL license. The program is available as a free and a pro version. The newer "PassMark" versions in general enable additional features. But they are only usable for systems with UEFI, for systems with legacy BIOS a user has to use older versions without this additional content. That fact has implications on the scope of the available features.

MemTest86 is a stand-alone software respective independent of the operating system, because it boots from an USB flash device. This works for Windows/mac OS or Linux.

All versions are capable of detecting faults in the RAM but also to determine RAM benchmarks as the memory speed and other properties. As mentioned only the newer versions enable to run a RAM benchmark test which delivers a plot. In this plot it is possible to see how the the actual memory bandwidth developments in dependency of memory read or write operations. The older versions deliver only one speed value for each memory/cache unit.

For the same system we calculated the max. memory bandwidth, we run MemTest86, too and got the following results:

- 8.770 GB/s

For the physical memory (size 8070 MB) and memory latency of 44.186 ns. Also important are the speeds of the different cache layers. In the case of our example

L1: 34.600 GB/s (size 4x64 KB)

L2: 18.390 GB/s (size 4x256 KB)

L3: 12.323 GB/s (size 3072 KB)

That lead to plot 2.2 where you can see the effect of the cache and after the blocks got bigger than cache, to the measured value. For read operations the values are slightly faster, for 256 KB we see for write a value between 9000-10000 MB/s, but for read at the range of 12000 MB/s. That behavior was the same for several measurements, at least if the size was bigger than the biggest cache.

This is only a example, but visualize that MemTest86 is a tool which helps to examine the gap between reality and the calculated maximum. This knowledge is essential for the calibration of our model approach.

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

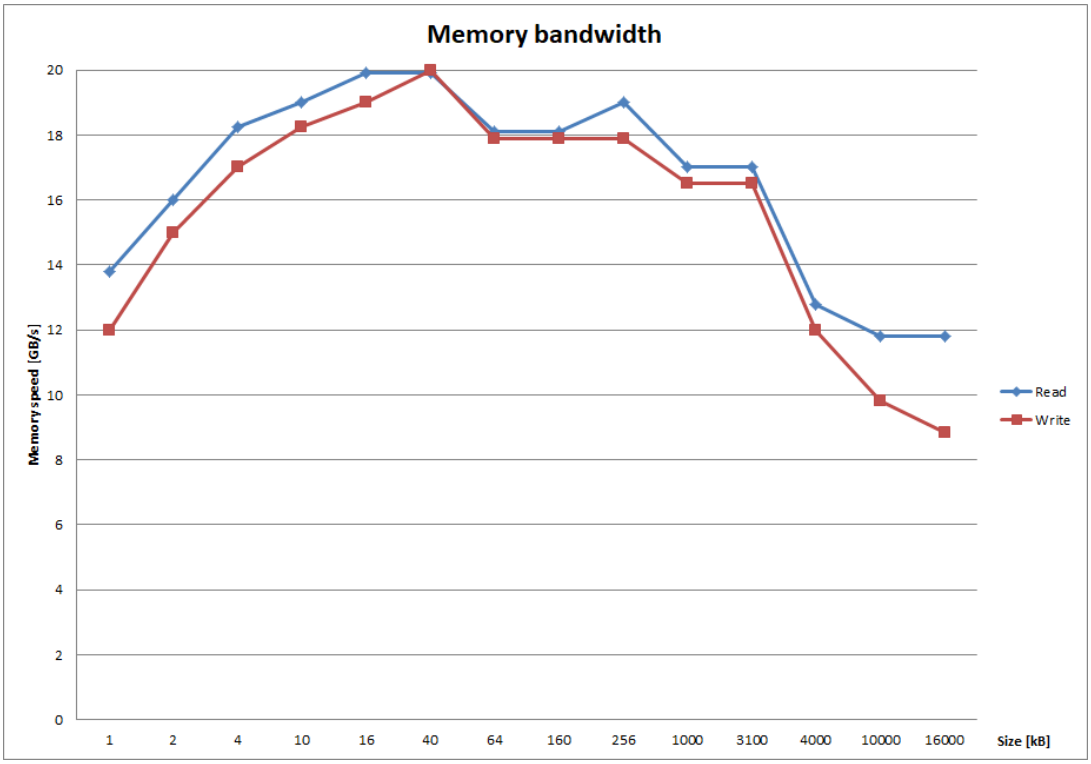

Figure 2.2.: Diagram of MemTest86 RAM Benchmark test results.

2.2.2. STREAM

The STREAM Benchmark was developed by John McCalpin and is a "de facto standard" regarding memory bandwidth. The rough concept behind this benchmark is to measure the sustainable memory bandwidth in MB/s. This measurement is done by four long vector operations: Copy, scale, sum and triad [McC95].

For the use in a uniprocessor case the C or Fortran file has to be compiled and the executable is ready to execute. But for multiprocessor systems they state on their website a few hurdles. For that case the code has to be run parallel. In order to do that the user should look up how that works on the used system. The code includes OpenMP directives, which allow a straight forward specification of the used threads. In addition the user should keep an eye on the cache size, because if the data is small enough to fit into the cache, the result will refer to the cache speed. Although the purpose of the STREAM benchmark is to measure the memory bandwidth of the main memory it can also be used for cache. Further information can be found on [McC07] or in the code itself. There it says: "Each array must be at least 4 times the size of the available cache memory". According to that the "STREAM_ARRAY_SIZE" should be changed. In the multiprocessor case the summed up cache size has to be taken into account. In theory

it is also possible to determine the cache speed with STREAM, but for such a purpose McCalpin developed STREAM2.

After running the executable the prompted result will look like this example:

```
Function Best Rate MB/s Avg time Min time Max time
Copy:        4363.4  0.036724  0.036669  0.036786
Scale:       4454.4  0.035960  0.035920  0.035988
Add:         5154.7  0.046619  0.046560  0.046642
Triad:       5025.5  0.047791  0.047756  0.047825
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
```

Listing 2.1: STREAM Result.

For our case MemTest86 seems to fit better, because it is more detailed and it is easier to achieve the desired results. In addition [WWP09] state that there would be adjustments needed to get the memory bandwidth. This is maybe not the most important issue for us, because their critics focused on the purpose for their work. More about that in the chapter 3 – Related work. In addition with the other factors this is clearly a disadvantage of the STREAM benchmark. Still it is wise to have an alternative, if Memtest86 fails we still could use STREAM.

2.3. Hardware performance counter

As this work distinguishes between the parts of the memory hierarchies, at some point it is important to know how the ratio between cache and main memory looks like. That means how much cache accesses results in cache hits or in cache misses respectively main memory accesses. The latter would require a mechanism which counts the cache hits and misses.

On PC´s with Windows or Linux OS, the task manager or htop are well known programs which are able to visualize some process, system and hardware informations. These both are not enough for our purpose, because neither htop or the task manager offer detailed cache/memory information for one specific program. However, a wide range of programs exists as for example the resource monitor or the performance monitor at windows systems or perf [3] for Linux. And these are only a few of many more, with similar capabilities.

[3] http://man7.org/linux/man-pages/man1/perf.1.html

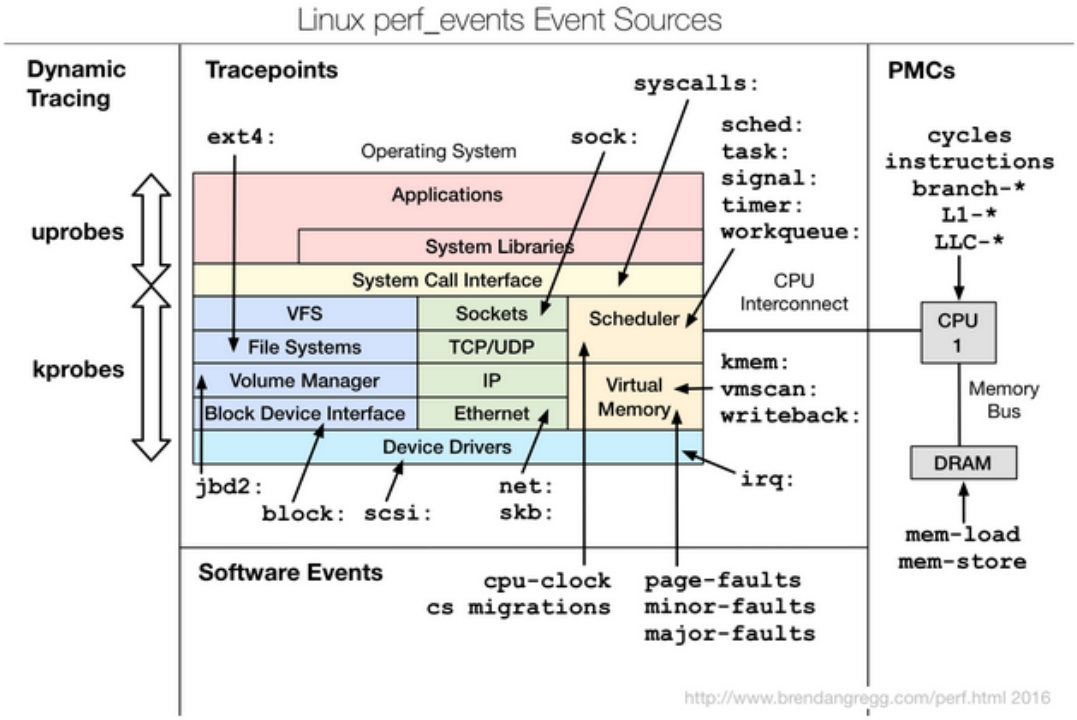

Figure 2.3.: Scope of perf, picture from [Gre19].

In order to chose the fitting tool it is important to understand for which purpose we need it. The target of this thesis is to calibrate our model with the right values. That means in the first place the speed of the different layers of the memory hierarchies. But if we have to decide their impact respectively how they are used by guessing, this is inaccurate. A hardware performance counter which would deliver the exact amount of cache access and foremost cache misses over the course of a program execution, is what we looking for. Perf or perf_events is a diagnostic for Linux which offers the functionality we strive for. It is included in the Linux kernel, easy to install and to use. For a documentation see their wiki or that what Brendan Gregg wrote [Gre19].

2.3 shows the capabilities of perf. Regarding the cache and memory, the left part of the picture is from interest for us. PMCs means performance monitoring counters and covers hardware events. Especially the counters for last level caches (LLC) and L1 are interesting for us. Statistics for them can be obtained by the following command.

```
perf stat -d
```

Listing 2.2: Command line call.

"Stat" enables to count and the d flag activates detailed statistics for a program call, which is placed after 2.2. For example if we run a Java program, we place perf in front

of the command line call. After the program is finished the results were printed. That looks like shown in 2.3.

```
perf stat -d java de/tuchemnitz/se/openmp/Main

140592.040154   task-clock (msec)  #   2.009 CPUs utilized
12,696    context-switches   #   0.090 K/sec
3,897       cpu-migrations     #   0.028 K/sec
231,257      page-faults        #   0.002 M/sec
333,854,585,149 cycles             #   2.375 GHz              (40.10%)
112,342,182,600 stalled-cycles-frontend # 33.65% frontend cycles idle (40.21%)
57,975,151,271  stalled-cycles-backend # 17.37% backend cycles idle (40.25%)
333,866,684,903 instructions       #   1.00 insns per cycle
                                   #   0.34 stalled cycles per insn (50.25%)
21,659,147,412  branches           # 154.057 M/sec            (50.22%)
45,129,664      branch-misses      #   0.21% of all branches  (50.15%)
247,148,555,856 L1-dcache-loads    # 1757.913 M/sec           (50.08%)
5,429,874,086   L1-dcache-load-misses # 2.20% of all L1-dcache hits (50.03%)
558,760,603     LLC-loads          #   3.974 M/sec            (40.00%)
435,441,008     LLC-load-misses    #  77.93% of all LL-cache hits (40.03%)

69.966024032 seconds time elapsed
```

Listing 2.3: A example result when using the command 2.2.

The last four points are related to the cache/main memory. With this information we are able to calculated the number of L2 loads and misses as well, which means it provides the whole information about the memory hierarchies. With one limitation: the numbers focus an load operations only, that means we have no information about the write operations.

2.4. Performance predictions

We already stated that software quality prediction models and tools help to prevent respectively detect poor design decisions. They enable to estimate the effect of design decisions and evaluate alternatives against each other, which is important as errors at this stage are expensive. [Bec08] (page 59 ff.) delivers a detailed description about performance prediction and their methods. After them it is important to take into account the component-based side as well as the model driven aspects. Hence they came up with three main requirements to combine these areas. First to support a component based development process, which means to consider the different roles in the development process, as they call them component developer, assembler, deployer and domain expert. Second, to benefit from the model driven software development

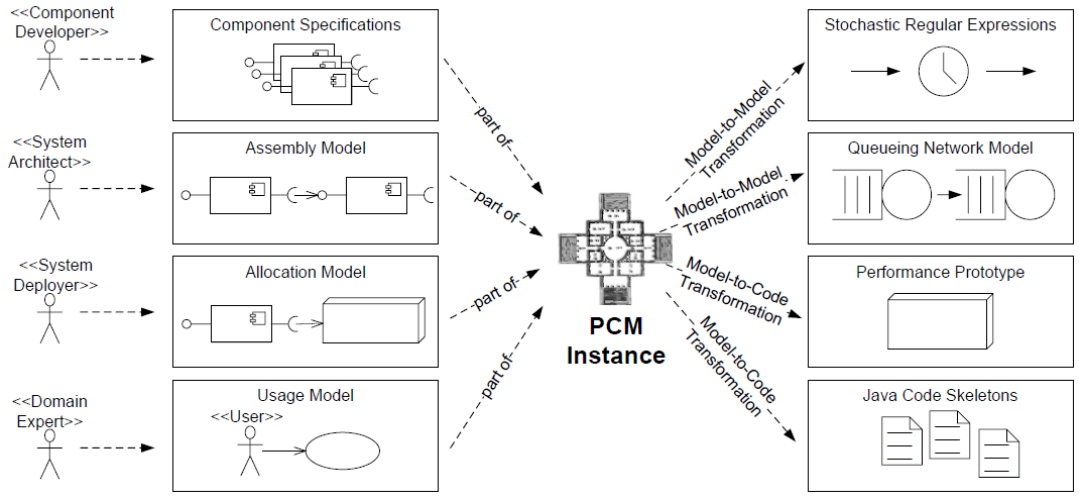

Figure 2.4.: Visualization of the PCM process [BKR09].

this should be also brought in as models and model transformation. Plus third, model driven predictions of the architecture should be considered.

2.4.1. Palladio component model

One solution for a model driven but also component based performance prediction is as for example Palladio respective the meta model PCM. 2.4 shows the mentioned separation of roles as well as the model and model transformation.

A Palladio project normally consists of an repository, system, allocation, resource environment model and usage. We proceed step by step in top down order.

Repository

Normally the entry point is the repository model, the figure 2.4 shows that the component developer is responsible for that. And that is exactly what is specified here, components which use provided interfaces and their method (signature) with their input and return parameters or/and components which require certain interfaces and their methods. It is also possible to model more fine grained components as infrastructure, composite components or sub-systems.

As the methodology suggests the behavior of the methods has to be specified, which happens in each component as SEFF (**S**ervice **Eff**ect Specification) or more precisely as RD-SEFF (**R**esource **D**emand **S**ervice **Eff**ect Specification) [BKR09]. They enable the

possibility to model internal actions with resource demands (e.g. CPU, HDD, Delay) or call external methods which were required by the component. There is also a possibility to use passive resources to simulate for example a thread pool or to use variables (usage/characterization) for resource demands, loops or network traffic. In addition it is possible to parallelize via fork actions or to build branchings with branch actions, which are able to delegate by condition or probability. The difference between fork and branch is, that a fork is only usable with a synchronization point. That synchronization point will arrange that the different "branches" will be synchronized at the end of their execution. [FHB18] states the SEFF has similarities to an activity diagram of UML2 and the repository to a component diagram.

System

The next step would be to specify the system, which actually means the assembly model in the diagram 2.4. In a more stricter sense here the hierarchies of the called components. That means which components provides the initial entry interface and how the call is then distributed. It is also possible that the system required another system, where the same happens again. In simple words here the relationship between the different components can be modeled.

Resource environment

Here the modulation of the hardware takes place. That means a resource container can represent a server, computer, database system etc. ([FHB18] call it compute nodes) with their respective capabilities (processing resource specification) regarding CPU, HDD, Delay: cores or replicas, processing rate, scheduling, mean time to repair (MTTR) and mean time to failure (MTTF). And how these resources are connected by a linking resource with latency, throughput and failure probability or not.

Allocation

The allocation model puts the system and resource environment together. That means the simple task here is just to allocate the components from the system model to a resource container where it should run. [FHB18] writes it has similarities to a UML2 deployment diagram.

Usage

Last point of this hierarchies considers as the title suggest how the system is used by the user. Therefore the domain expert specifies how and which interface from the system will be called. This happens by a so called "Usage Scenario", which has some redundancy to the SEFF because the possibility of branching, loops, variable usage/characterization and entry system calls also exist. Unique are the delay and the workloads. In this workload it is possible to model how much users (population) use the system and with which amount of think time or in another case the usage intervals.

This is only a basic description of Palladio and its capabilities, the newer Sirius editors provide even more as for example the AT (architectural templates) [Leh18] [FHB18]. The AT offer basically the opportunity to reuse or specify well known architectures as for example a load balancer. That provides a potential possibility to save modeling effort, because the AT creates automatically specified parts, for example a load balancer for a number of components.

Chapter 3

Related work

This chapter treats approaches which had a similar target or a different but at least a similar result to our work. In addition we explain the work from [FH16] in more detail, because it is the baseline for this work.

3.1. Multicore performance models

The work of Williams et al. [WWP09] had the target to introduce a performance model which is easy to understand by programmers and should help to provide insights into the behavior of multicore systems. This is valuable because once the multicore processors were new introduced, there was a lot of uncertainty and nothing like "conventional wisdom". That term means that for single core processor a common sense exists in terms of design/architecture or use regarding performance guidelines.

They mentioned the 3C´s model as an example for such a performance model, which is basically a model for CPU caches with the named properties. Williams et al. [WWP09] treat in their model the correlation between transferred bytes through the DRAM and floating point operations with a so called roofline, which is the upper bound regarding the performance of a kernel. This upper bound is depending on their introduced term *operational intensity*, which means mean operations per byte of the DRAM traffic. The DRAM traffic is the traffic that was already "filtered" by the cache hierarchy and goes into the DRAM. That is a difference to other definitions as for example machine balance, which is according to them treating the traffic between processor and cache. And they did not want to exclude kernels that only work with non arithmetic operations.

This allows to determine, if the upper bound of a kernel is related to the memory (bandwidth) or to the computing (floating point performance). The point where the increasing peak memory bandwidth does not increase anymore and intersects the peak

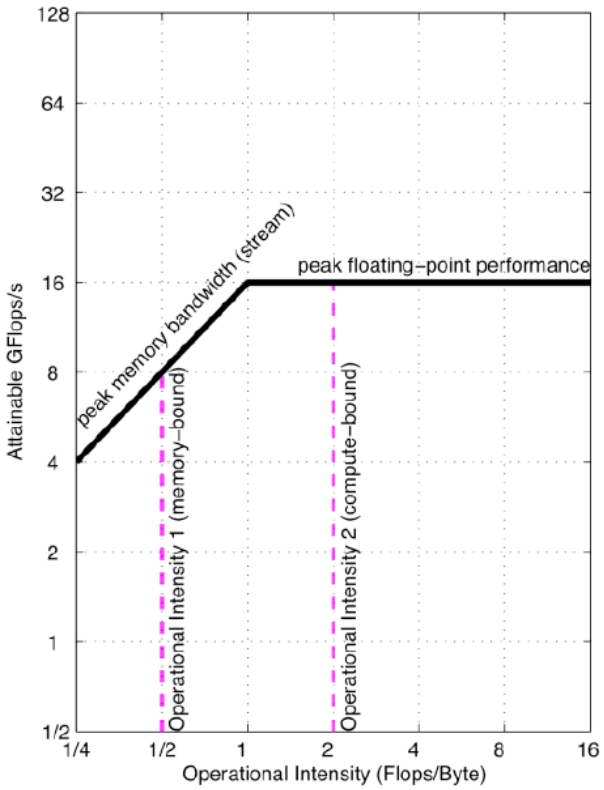

Figure 3.1.: Plot from [WWP09] to visualize the roofline.

Listing 3.1 Formula to get the roofline.

```
Attainable GFlops/sec = Min(Peak Floating Point Performance, Peak Memory Bandwidth x
    Operational Intensity)
```

floating point performance is the course of the roofline or the ridge. This ridge allows to judge about the overall performance of the processor. 3.1 visualizes a roofline example, on the y-axis attainable/achievable GFlops and on the x-axis the described operational intensity.

The formula 3.1 shows how the "roofline" is determined. The roofline is also called Attainable GFlops per second, which means in different words the performance limit (peak) of the system.

In addition, the exact position of the ridge point, e.g. far right or more in the center/left, allows a suggestion how difficult it is to achieve the maximum performance. The roofline is bound to the "hardware/computer", it stays the same for all different kernels or operating systems. In order to do this the x-coordinate of the ridge point is the minimum operational intensity to reach the peak performance. This allows to compare

different processors and how much operational intensity is needed to get a potential higher "peak" performance.

Since the roofline is the upper bound the authors were aware of that problem that programs could perform below the roofline. In the first step we mentioned that the roofline enables to spot if the "bottleneck" is related to the memory bandwidth or the computing performance. In addition to that they suggest optimizing measures for both cases. For the matter the term "ceiling" was introduced, as an intermediate bound to get to the roofline. With these ceilings it is possible to visualize the bound of different states before/after optimizations, regardless if for the computing or bandwidth part. The operational intensity tells on which optimizations a user should focus. For example if memory bottlenecks are the problem they suggest: ensure memory affinity or the use of software prefetching. These measures could help to metaphoric break through the ceiling. For this the paper presents a detailed manual with recommendations and reports from their experiments with four different processors. One limitation of the operational intensity is that it is not fixed. The value can increase with the problem size. Furthermore optimizations regarding the cache influence the operational intensity. That is why they included the 3C´s model into the roofline model.

Moreover, they showed that it is not simple to reach the peak performance and even with optimizations it is probably still far off. In contrast to their experiments the processor with the lowest ridge point was the easiest to optimize rather than the processor with highest ridge point. The latter was very complex to optimize and had the lowest unoptimized performance of all viewed processors. Another observation was that the majority of the processors was memory bound.

The authors concluded that the ridge point of their roofline function was a better performance predictor than for example the clock rate or the peak performance. Although they had another target plus their work is from 2008 and therefore already a bit outdated, it seems still relevant for us. Another similarity with our work is that they measured the memory bandwidth to calibrate their model. They did that with the STREAM benchmark and claim that the default program does not measure the main memory bandwidth. STREAM measures the iterations per second as performance and then deviates the memory bandwidth "based on the compulsory memory traffic on a non-write allocate architecture." That was problematic for their work because this did not include conflict misses or write misses. Therefore the authors adapted STREAM to get four bandwidths with different settings. Our thesis has not such a specific use case, therefore this seems less important for us. But this is one additional argument to use MemTest86.

[VE11] also made an attempt to present a multicore performance model: The multi-program performance model (MMPM). The model is based on single core simulation runs and then switches the focus on several programs which were running concurrently. Or as they call it a representative workload, something that in accordance to them is not easy to achieve. Due to the fact that it is unfeasible to simulate all possible combinations of workload "mixes" because that would take too much time. The alternative was to pick only a limited number of random workloads. But this solution leaves doubts, if these workloads are representative. That is the issue where van Craeynest and Eeckhout want to introduce their model, for the sake of a better solution with "less" doubts of being representative. In addition their simulation duration is relatively fast.

The model works as follows: the single core performance is the input for the model. From then, the input goes through several iterations with the target to simulate how shared resources affect the performance "per-core" for multi core systems. The input or the single core profiles is consisting of three characteristics: Single Core Cycles per Instructions (CPI), Memory CPI and Stack Distance Counters (SDCs).

The Single Core CPI is the number of CPI when running a workload, in detail it is the number of cycles count divided by the number of dynamically executed instructions. The Memory CPI is the part of the Single Core CPI which is waiting for memory. There are two ways to compute, one is to run two simulation runs, one with a perfect last level cache (LLC), which means all accesses are hits. The second run is with an imperfect last level cache, without any hits. After that the latter is subtracted from the first value to get the memory CPI.

The Stack Distance Counters (SDCs) reflects the temporal memory access behavior of a program in caches. It is calculated by incrementing a counter, when the cache is accessed. More precisely the counter for the specific element/position of the LRU stack which is successfully hit will be incremented. That means an A-associative cache has A + 1 counters. The plus one is because the cache miss case has his own counter, which will be incremented for that. Those "profiles" were measured in intervals for every 20 million instructions. After this step, the model proceeds by iteratively converging to the affected per core performance in multi core systems. The performance is affected by resource contention in shared resources. Both factors, resource contention and per core performance are attached to each other in both directions.

In detail the schema starts by defining a variable R_P. This is the slowdown for a program in the multi core context, which is one at the entry point. This is the same with the execution trace pointer, which starts with the value zero. In the next step the program P with highest multi core CPI is determined for the next interval of L instructions, then the C_P is calculated by multiplying the single core CPI with the slowdown factor R_P and L the instructions. The variable C is also determined at that step as cycles of the slowest program. The next step defines N_P as progress of the program for the next cycle. Then

the SDCs should be computed for the N_P of instructions. The SDCs are used as the input for their used cache contention model in regard to estimate the number of cache misses. The cache contention model of van Craeynest and Eeckhout is the Frequency of Access (FOA) from Chandra et al. [CGKS05]. This model is also a performance model that investigates the impact of cache sharing and propose. The output of this model is the number of extra L2 cache misses because of the cache sharing. Secondly, the SDC helps to estimate the penalty for an LLC miss. This value is calculated by dividing the number of lost cycles because waiting for the memory (Memory CPI $*$ number of instructions for the interval) with the number of LLC misses (number of the counter, if cache is not hit).

This is one iteration of the slow down factor calculation until the stop criterion is reached.

To evaluate their model they used three metrics, the System throughput (STP), the average turnaround time (ANTT) and the predicted versus the measured per-program slowdown. The accuracy, respectively the average error was on their set up with the SPEC CPU2006 and the use of up to 16 cores, between 2.3% and 2.9% for STP and ANTT. For the predicted versus the measured per-program slowdown it was up to 4.5%.

This work is also interesting for use, because they dealt with the calibration issue as well but more detailed than the roofline. In this case van Craeynest and Eeckhout rather focused on factors which limit the performance than on the whole difference between a single core to a multi core processor. A limitation is that they do not consider shared bandwidth or prefetching. However, in terms of shared cache behavior the model seems helpful to give an impression on how such a model works and how accurate they are.

A third performance model is CAMP from Xu et al. [XCDM10]. The model targets the resource contention in shared caches as well and offers according to the authors a fast and accurate estimation of slow down effects due to that matter. The motivation for their model was the observation that the impact of cache contention on the performance is application dependent. Therefore they effort a model which takes the cache access and contention in consideration.

This is done by defining the following parameters: The average number of cache accesses per second (APS) calculated from the cache accesses per instruction (API - a process property) divided by seconds per instruction (SPI). The SPI has two parts, firstly the computing (on-chip) latency and secondly the (off-chip) latency due memory/disk

accesses which includes cache misses (MPS). The on-chip latencies are constant, if the CPU frequency remains constant too, they calculated the SPI as followed:

(3.1) $SPI = \alpha * MPA + \beta$

The MPA specifies the last level cache misses per cache access. The amount of the APS allows assumptions on how processes compete for the shared cache. This can be calculated by:

(3.2) $MPA_i(S_i) = \int_{\infty}^{S_i} hist_i(x)\,dx$

Note that i variable represents the process, S_i the cache size, while $hist$ is a continuous function derived using linear interpolation of support estimation for non-integer average reuse distances. The parameters α and β are obtained during the offline characterization. Furthermore this is only the performance model, later they try to estimate the effective cache size after n accesses and what are the conditions to reach a steady state. Which is also complex but there is an alternative - automated profiling. This is possible to use hardware performance counters (HPCs).

That enables to collect SPI and MPA accurately but not regarding the reuse histogram data. In addition they set up a little program called stressmark, which is helping them to do that, also for the parameters of a specific process. This is also the point where the α and β from above can be determined by linear regression. For more details see their paper [XCDM10].

So far the theory, in their validation they used a "Intel Core 2 Duo P8600 processor and the Mac OS X 10.5." First they run their stressmark to determine, then in the next step they used ten benchmarks from the SPEC CPU2000 and analyzed the results in comparison to the predicted results.

Their average prediction error was at 1.57%, which is better than the model proposed by Craeynest and Eeckhout. But they only tested it in a two core scenario other than Craeynest et Eeckhout who scaled up to 16 cores. Still CAMP delivers interesting insight in how to calibrate a performance prediction, and it seems their approach is also a bit less complex because they use some sort of automation. The use of hardware performance counters for example is an interesting approach.

3.2. Work to extend

In this thesis we rely on preliminary work and target to extend this work. Therefore we present a short overview about it.

Listing 3.2 Code of the matrix multiplication.

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

[FH16] delivered preliminary work and pointed out how ready Palladio is for multicore systems. They showed it within an experiment. For the implementation of the matrix multiplication in Java they used OpenMP. OpenMP is a model/API which simplifies the parallelization of sequential programs. The matrix multiplication was parallelized for 2, 4, 8 and 16 threads. Both matrix sizes were 3000x3000 and filled with random integer numbers. The calculation was repeated 500 times, to prevent optimization the matrices were filled in each run again with random numbers.

The used approach to use OpenMP in Java was the omp4j [1] framework. It should enable the parallelization by specifying the number of threads/cores in a comment or more precisely a compiler directive above the code part which should be parallelized. Just like this

```
// omp parallel for schedule(static) threadNum(2)
```

where the number of cores was two. The execution time was measured around that code snippet.

In order to show that it is possible to model such multicore systems with Palladio, they had to bring the OpenMP model into Palladio.

They realized this as a "workaround" because Palladio only supports basic concept parallelization by default and they had problems to realize it with passive resources, because a bug leads to an exception if passive resources were used in a fork action. This means it is not possible to model a "thread pool" and just adjust for the number of cores the thread pool size/loop runs. Instead they modeled each worker thread in the SEFF as a separate branch in a fork action. A fork action in the PCM context means, that the forked actions respectively the actions inside of the fork action will be executed concurrently. In addition the control flow will not proceed until the "forked behavior is terminated" [BKR09].

Then they scaled up the internal actions with the available cores, which also means that they adapted the resource demand. The resource consists of the three matrix sizes which were the exit condition for the three loops in the code from 3.2 and a calibration. On their hardware a calculation of two values took 0.69 nanoseconds, see A.2 in the

[1] http://omp4j.org/

appendix. 18.64 sec is the mean time the system took for the multiplication with a single thread/core A.2. The experiment was run on a 16 core machine (2 CPUs a 8 ores with 2,4 GHz and 20 MB cache). That means this and the sizes were multiplied, divided by the number of cores to simulate the CPU demand.

In the Palladio itself the value is not set in nanoseconds, they used microseconds. This is possible because Palladio is most of the times without a specific unit. But if it comes to the response time, the csv or the diagrams suggest the values are in seconds. In this case this is false, which means the value is actually in milliseconds.

It was also not possible to model the resource demand into a loop action, which iterates $3000 * 3000 * 3000$ times, because the number got too large for Java. Therefore they opted to multiply the calibration 0.69 with the number of loop iterations in one resource demand per core, divided by the used cores. In the last step they increased in the resource environment the replicas of the resource container by the number of cores. In the multicore case the Exact Schedulers from [Hap09] was used, because the "Linux 2.6 O(1)" takes context switches into account. They occur in the multicore case, for the serial model it was enough to use processor sharing. The exact scheduler also causes problems in conjunction with the passive resources.

All of that costs a lot of effort in modeling, especially with the increasing number of cores. Since that means for e.g. 16 cores, it is afforded to model sixteen internal actions in the fork action. Therefore the usability is cumbersome and error prone. In addition Palladio offers sometimes non-specific error prompts. That often leads to time consuming research to find for example a typo.

Furthermore, the accuracy is numbered at 79% (comparison between simulation and measurements of a matrix multiplication, implemented with the help of OpenMP). There is room for improvement, especially the accuracy decreases for an increasing number of cores A.2. This is the point for continuation and realizing their suggestion for future work, evaluation and consideration of memory bandwidth, cache and size. As they suspected costly synchronization of cache with the memory is one factor for that gap.

Chapter 4

Approach to address shortcomings

In this chapter our approach to address the shortcoming of PCM in the multicore context will be described. Basically this is a detailed description on how existing elements of PCM can be used to model memory bandwidth and the memory hierarchies. PCM offers Resource Types to model resources [BKR09]. First the ProcessingResourceTypes (e.g. CPU, HDD), second the PassiveResourceTypes (e.g. semaphores) and third CommunicationLinkResources (e.g. network connection), which is a specialized ProcessingResourceType. The last two are interesting for us and will be described in the following. After that we will explain how we want to use/calibrate them.

4.1. Network links

A network link in Palladio is a processing element or in other words an active resource, because it does some sort of processing and is able to execute a task "independent" [RBK+11]. It allows to connect different resource containers with each other. 5.3 shows an example modeled in Palladio Bench. A network link element provides the opportunity to set the throughput, latency and failure probability of the connection. The throughput is the value how much traffic the network can handle over time, for example of one byte goes through and the throughput is also one byte/sec, than one second will added for this process for the network demand. The latency is the round trip latency [Hap09], which means the time it takes to go from one resource container forward and backwards.

This was only the network link representation in the resource environment model. A network link has to specify a network demand. This happens in the repository model or more precisely in the SEFF respectively Resource Demand SEFF (RD-SEFF). First of all two different components are required, among them the network link or connection

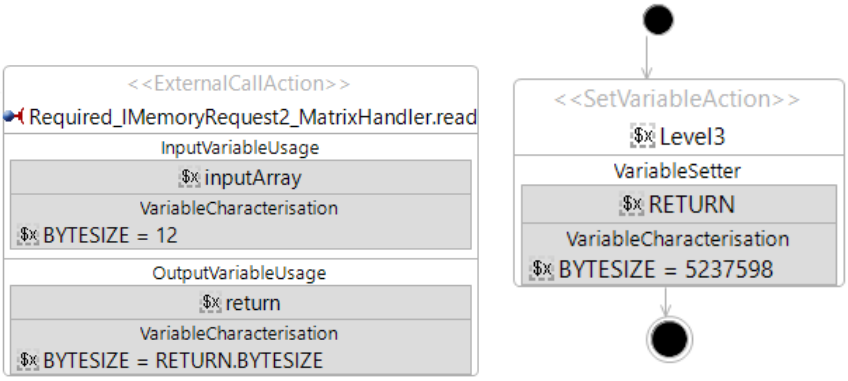

Figure 4.1.: On the left the externalCallAction from the trigger component is pictured, on the right the VariableAction from the called component.

will be established. The two components should be allocated on two different resource containers, which should be linked in a way showed in 5.3.

The linked components then have also be connected in the repository model, because the network demand is triggered by an *ExternalCallAction*. The component which triggers the *ExternalCallAction* has to require the interface of the component which is provided by the second component. In addition this interface has to specify a so called "network demand" method. This method needs to specify a return type and an input parameter. Without specifying them it is not possible to proceed. In the SEFF of the component which triggers the network demand, the external action is placed as shown for example in 4.1. In the invoked component a *SetVariableAction* has to be placed, showed in 4.1 too. The graphic shows also a variable usage with a VariableCharacterization. The usage defines the variable or the variables which are transfered over the network. The characterization sets the size of the transfered data. In general Palladio offers for a variable characterization five different abstraction of parameters: VALUE, STRUCTURE, NUMBER OF ELEMENTS, TYPE and BYTESIZE [Bec08]. Only the BYTESIZE parameter type is able to determine a network load. The size of this value determines the actual number of data which is send over the network by the issued call.

With that set the network demand is ready to be run in a simulation/experiment. Before running the experiment for the first time it is needed to set up the run configuration. In case of using a network demand, the "Feature Settings" tab offers a additional possibility to change the network behavior. It offers three possibilities: Simulate throughput of remote calls, simulate full middleware marshalling/ demarshalling of remote calls and do not simulate throughput of remote calls. The third option makes the changes described above useless, as there will be no network demand.

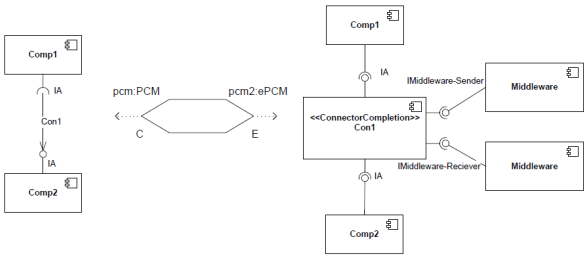

Figure 4.2.: Figure from [Bec08] to illustrate the changes at the connector.

The difference between the other two options is that connector will be replaced. A visualization with a middleware element is showed in 4.2. That option enables to define additional elements for the network process and to specify their performance impact. [Bec08] page 181 ff. lists for example different protocols (e.g. RMI or SOAP) and additional processing (encrypting, compression, authorization, decryption, etc.) which then would executed on a middleware "container" between both components or resource containers. Palladio has default specification for a Java EE Application Server (GlassFish [1]) determined in the "Glassfish.repository". It includes additional processing: encrypt, decrypt, masrshall, demarshall, create credentials and check credentials. This can be changed see [2]. [BKR09] stated: "In the future, we will extend this simple model with special modeling constructs for middleware (such as caches)." But until now such an element is not implemented.

4.2. Passive resources

Other than active resources, passive resources are not "independent", because they are only usable in conjunction with a process. This process or thread uses the passive resource for a certain time period. But before the process is able to do that, it has to acquire one of the limited resources. After the process is finished it has to release the resource. The resource is then free to use for other processes/threads. That is a similar concept than used by thread pools [RBK+11].

In PCM the passive resources are part of the repository model ore more precisely of a component. They are not mandatory but if desired every component can consist of one or more, with a specific number of capacity in the PassiveResourceCompartment, see 4.3. The specification enables the passive resources to be used in the RDSEFF, with acquire

[1] https://javaee.github.io/firstcup/
[2] https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model/FAQ

and release actions [BKR09]. Those actions are based on the semantics of a semaphore [RBK+11]. For example a resource demand can enclosed with both, which means the demand can only be issued, if he can acquire a token from the limited capacity and if he is finished he will release it back to the resource pool. In this case it is possible that a process has to wait for the passive resource, because none is available to proceed. As for example a database connection pictured in 4.4. The capacity of 15 limits how often readData internalAction can executed concurrently.

The passive resource could be useful to model synchronization/overhead and cache contention. These effects are also some sort of a race for a limited resource.

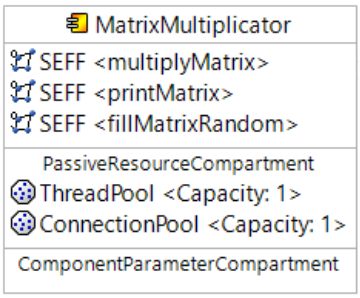

Figure 4.3.: Screenshot from Palladio Bench to illustrate how to specify a passive resource.

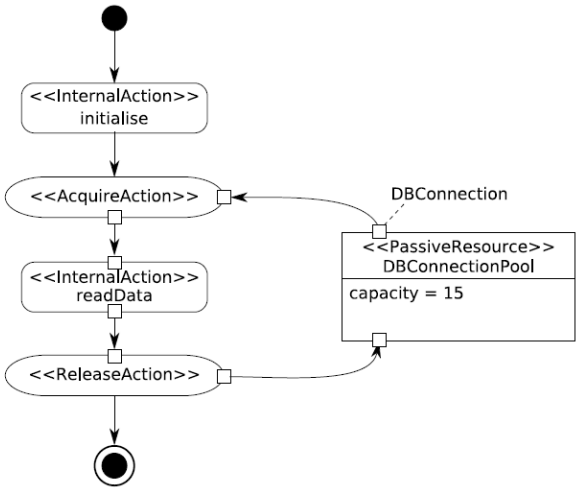

Figure 4.4.: Figure from [RBK+11] to illustrate the concept of passive resources.

4.3. Calibration

In the first place calibration means gather the data from the Memtest86 run and plug the speed values into the Palladio elements. We learned above that it is possible to specify the throughput and the latency of a network connection in Palladio. A bus or the memory bus has conceptual similarities with a network or LAN, but both differ in specifications/properties regarding performance. Hence it is enough to adapt the values of the Palladio network links, in order to get a realist representation of the memory bus or the cache hierarchies.

Supposed that Memtest86 determines a "memory bandwidth of 23.976 GB/s, this would be the throughput from the CPU container to the RAM container. This is the same for a memory latency of 23 nanoseconds but now as latency of the network link. Although we have consider that network link uses is as round trip latency. A workaround could be to divide the latency value by two. In almost the same manner we could proceed for the different cache levels. That is a coarse grained use case because the detailed Memtest86, see 2.2, gathers data that distinguishes between write/read operations through a reasonable margin of data size. Hence we could model our use-case as fine coarse as our data. The trade-off between regarding benefit of getting more accurate results and modeling effort, decides if this makes sense or not.

But this is only one part of the calibration. In the 3.2 we described in detail the use-case we aim to remodel or to extend. Therefore it is essential to know how much data is going over the memory bus for this matrix multiplication. We have two possibilities to get this knowledge, firstly by measuring it or secondly by calculating it. We decided to calculate it because this seemed the straight forward approach and easier to execute.

The calculation looks as follows:

First of all we needed to know the size of an single element that flows over the memory bus. The matrix multiplication is written in Java. An Integer in Java is a 32 bit signed value see [3]. That means a Java Integer has a size of 4 byte. This is the size of an elements or a number of the matrix multiplication that will go through the memory bus. We also know that the experiment multiples two 3000x3000 matrices.

One 3000x3000 matrix with Java int values has the following size:

- $3000 * 3000 * 4 = 0.036$ GB

Due to two as an input before the multiplication is able to start. 0.072 GB were loaded in the memory and after the multiplication is finished 0.036 GB will be written back as

[3] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

final result. But this is not meant by the actual memory bandwidth. For this we take a look at the algorithm of the matrix multiplication of the section 3.2, see listing 3.2. We know already that the width respectively the height of the matrices is 3000. Therefore the three loops will iterate:

- 3000*3000*3000 = 27 000 000 000 times

So the actual multiplication step will be executed that often and produces per run three memory reads and one write after getting the intermediate result. To keep it simple we assume a four memory accesses for each calculation step, and with taking their size into account, we have:

- 4*4 = 16 byte

data transfer to the memory with each loop iteration. Overall that means:

- 27 000 000 000 * 16 Byte = 432 GB

data transfer. This is the same regardless of the number of cores respectively degree of parallelization. But on factor changes with the degree of parallelization: The time distribution of the data transfer.

[FH16] measured 18.64 sec for a single core matrix multiplication. In this case the 432 GB data transfer means 23.5 GB per second. This value increases simultaneously with the number of cores. A linear speed up for a two core scenario would mean 9.32 sec response time. Then a throughput of 46.35 GB/s should be achieved. In the four core scenario the value doubles to 92.7 GB/s etc. This illustrates the bottleneck which is described by memory bandwidth. The Palladio performance simulations of the same multicore use-case is predicting pretty much a linear speed up, because their model takes this not into account. Our plan is to change that, what means the calibration should add for each loop iteration a network demand of 16 bytes respectively the mentioned values in a overall scenario. That should then slow down the "response time" by the additional network effort. For example with a throughput of 23.976 GB/s the two core scenario will produce roughly a second to send the data. This will double to two for four cores and so on, see A.1.

That is the theory, in reality the memory hierarchies play a role too. We learned before that caches are faster than DRAM. Furthermore the topology is: A processor is connected to his belonging private caches. Then with the inter-core shared caches and at the last level the DRAM (simplified). For the calibration this is significant, because in a ideal scenario we could measure the exact cache behavior together with the calibration measurement. Under the assumption that the cache behavior stays stable for an increasing number of cores. This assumption efforts a verification Another thought could be use a fixed ratio of 90:10 or similar like a "performance model" (see related work 3), which has proven to be accurate.

The complexity of the calibration could be an issue, if we try to bring all possible effects into our model. For example to distinguish between all operations, regardless of cache/DRAM, write/read. Hence, we aim to neglect the difference between write/read, because that speed gap seemed for all our measurements relatively small. The difference between the memory hierarchies in contrast is significant.

The calibration of our memory hierarchy models working the following way: We use megabyte (MB) for the throughput rate of the network elements, the cache/memory transfer will be represented in MB, too. But since the models of [FH16] return a response time in millisecond instead of second (e.g. 26458 as response time is equivalent to 26.485 sec), we have to add three digits/zeros to the data traffic to compensate this representation. Otherwise we would only add the value in the wrong unit. For example: A 10 000 MB throughput adds for a traffic of 1 000 MB a tenth or 0.1 sec to the response time. For 26458.0 as a response time in millisecond that means 26458.1, which would be wrong. With a compensated traffic, 1 000 000 MB instead of 1 000 MB, it would be 100 milliseconds which would be added to the response time. See:

- 26458 + 100 = 26558 –> 26.558 sec

That was the calibration we intended to implement in our Palladio models, specific deviations are described in the next chapter.

Chapter 5

Experiment

In this chapter the whole process with the three iterations is described in detail. The results are described in the next chapter, here we focus on the course of action.

5.1. Iteration 0: Prototype

In the chapter before we learned the characteristics, capabilities of the Palladio Network Links and how to use them. With the help of that the first step was to model a network demand between resource containers. It is possible to start a Palladio project by modeling the resource environment at first, in this case by two resource containers which are connected by a network link. After that the repository model has to be modeled next. Due to the fact we want to establish a connection between two resource containers, we need also two components, which are connect by a required relationship to enable a externalAction call and subsequently a network demand. In the same manner as it is pictured in 4.1.

Afterwards we observed that changes in the resource model require changes in the system and allocation model. The two components have to be allocated to the two different resource containers. Regarding the system model, both components have to be connected via an assemblyConnector. With that we were able to create a system of two resource container linked by Palladio networks links.

With this first prototype we were looking for a simple use-case to model the memory bandwidth and the cache hierarchies. The choice was in favor of modeling the Memtest86 use-case, just as it is pictured in 2.2. The advantage was that we conducted the Memtest in either way to model any use-case, which was influenced by the memory bandwidth. Since the fact that we needed the memory and cache speeds of the executing machine. Hence, we added step by step one level of the memory hierarchies to our

model or in other words: for every cache or DRAM element its own resource container linked by its own network link element to a "fourth" resource container. This fourth resource container had the role of the CPU, although it would be more realistic to model a chain. But it brought no benefit for our result, it would just more complicated to model it in the other models. The network link elements were filled by the corresponding throughput/speed and latency values. Each resource container has its own component and RD-SEFF and variable declaration. In the variable declaration we set the byte size which corresponded to the size of the cache respectively DRAM element. All memory/-cache components were required by the "CPU" component, which then triggered the external calls. This behavior is also reflected in the system model and as mentioned above, each component has to be allocated to his own AllocationContext.

By doing this it is possible to get a response time, how long it would take to conduct the "RAM Benchmark" of Memtest86, if we would assume that just the time to transfer the data is a factor. That means we are neglecting potential overhead and synchronization effects. It is also possible to model for each point in the plot its own resource container/link resource. That would increase the accuracy of the response time, similar as to distinguish between read and write, but we decided to neglect this, see 4.3.

In this phase we also tried to use the AT (Architectural Template). There was no fitting element which could have helped us for our purpose, this was the same for the marshalling/demarshalling option regarding the network behavior. With the result from above the prototype iteration was finished.

5.2. Iteration 1: Modeling

The first step of the modeling phases was to get an impression of the work to extend described in 3.2. The experiment repository [1] also includes Palladio models that could be easily imported into eclipse, if Palladio Bench is installed. The models were not instantly usable because during the course of time the Palladio version changed. The Palladio wiki [2] site offers under "Notes on migrating PCM Models from PCM 4.1.0" how to make them usable again. This worked for us but we still had problems to execute them. Therefore the remodeling of these Palladio models was inevitable. With this approach it was no problem to repeat the simulations and getting the same results.

The next step was to consider the integration of the Memtest model into the matrix multiplication. For this purpose we had to change the system, resource, allocation

[1] https://gitlab.hrz.tu-chemnitz.de/marfra–tu-chemnitz.de/ssp_ramBW.git
[2] https://sdqweb.ipd.kit.edu/wiki/PCM_4.1

and repository model. Except for the latter this was straight forward. Although we tried to split the resource containers instead of scaling up the replicas. The aim was to model the core to core (private cache) communication and synchronization. But this had no positive effect other than to create more complexity. In order to explain that, for the same manner we tried to split everything that was in a fork action in the MatrixMultiplicator into one component with the same internal action. Later we brought them together by fork action in the ExperimentHandler, which included the external calls of these components and their network demands. It was also planned to hand over the "input" parameter from the usage model. But this approach did not work as intended.

Therefore we made the decision that it would be reasonable to issue the ExternalCallAction in the MatrixMultiplicator. This meant we only changed the MatrixMultiplicator by adding parts, not by removing or replacing a single part. The multiplicator component required for that purpose the cache and memory components. In order to avoid any disturbance we created a new resource container for the ExperimentHandler. In the old model, the handler was allocated on the same resource container as the MatrixMultiplicator. The last question we had to answer was where we should place the network demand. The initial idea was to change the loop in the multiplicator in a way, that for each iteration a read/write or a cache/DRAM demand was triggered, see 5.1. But as [FH16] noticed this loop modeling leads to the result that the execution does not terminate.

The fact that it is not possible to use such a loop had the consequence that we were forced to model our network demand in one ExternalCallAction outside of the ForkAction. In order to keep the distribution of the memory bandwidth with an increasing number of cores, it was necessary to invent a workaround. Our plan was to use the calibration measurement of [FH16] (18.64 sec) and calculate the linear speedup for the used cores. Then we would divide the overall memory demand (432 GB) by that linear speedup. That value (see A.1) got bigger with the decreasing execution time of increasing cores. That value represented the issued network demand for the correspondent number of cores. In addition we experimented with different cache/memory ratios. In addition we had to change the latency, because the value represents the latency for one load/write operation not for a ExternalCallAction that issues the whole demand at once. The solution was multiply the latency by the amount of overall memory operations.

In the same situation we noticed that with the Sirius editor we were not able to place a ExternalCallAction into a ForkAction. This odd behavior lead to the fact that all our models had to be modeled with the old GMF editor. This is no problem at all, we could even mix up both editors in one projects without noticing an effect. In avoidance of any disturbance we strongly prevented this. It turned out to be the best to trigger the network demand for each component at once outside of the ForkAction. We checked if

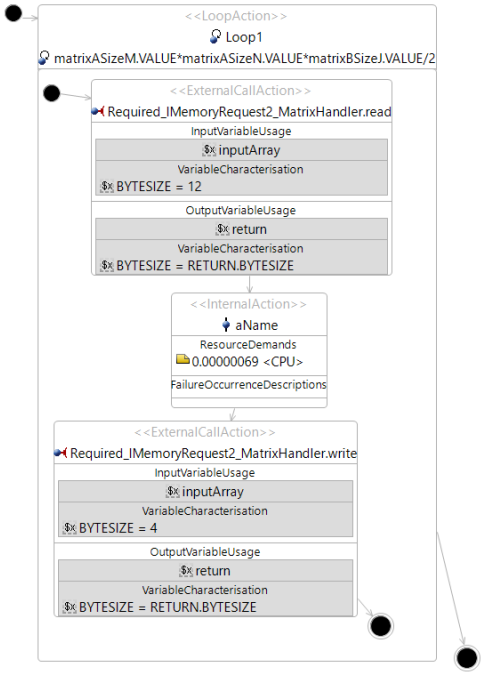

Figure 5.1.: Screenshot from Palladio Bench of the failed loop modeling.

the order of the external calls did change anything, for example to place them either completely after, before or around the fork action. This was not the case. The passive resources did not fit in our model, because the flaws of Palladio [FH16] described still exists. That means it is not possible to use the passive resources in combination with the Exact Scheduler or in a ForkAction. Therefore we skipped to include them in this model.

Another problem remained, we had limited information about used the processor in the existing experiment, neither specific model nor used RAM size etc. In addition the used hardware was not available for us. Therefore, it was not possible to initiate a Memtest86 RAM Benchmark in order to calibrate our models for their experiment.

This is one reason why another experiment was required, because the only other option to calibrate our model is to use the theoretical memory bandwidth for a comparable hardware with two CPU´s a 2.4 GHz and 8 cores. This would still be very inaccurate as we learned in our foundations chapter. Likely it would be better to use a Memtest86 measurement from the closest comparable hardware we got, which is far from ideal. A second point is our intermediate result, which showed an under estimation for two/four cores and an over estimation for the rest of the response time in comparisons to the real time measurements. This created an additional interest in measuring/observing the cache behavior. Apart from this our model seemed reasonable and easy to adapt if had to fine tune them. Therefore, we finished the modeling iteration.

5.3. Iteration 2: Experiment

For our experiment we had two machines with the following hardware:

- 32 GB RAM, 4 Intel® Xeon® Processor E5-2640 15M Cache, 2.50 GHz, in sum 24 cores
- 896 GB RAM, 8 Intel® Xeon® Processor E7-4870 30M Cache, 2.40 GHz, in sum 80 cores

Both machines had as OS the Ubuntu 18.04 and were accessed via ssh and putty. But this was not enough for us because putty only offers access to prompt of the machine. The Memtest86 cannot issued and observed via command line. Also we had the problem that we are not able to get into the boot menu via ssh. As a consequence we planned to set up an VM, we tried VMWare and VirtualBox. The latter was easier to install. Back to the command line issue, to solve that we installed x2go server on both machines and in order to get a desktop environment XFCE. In combination both enable to view and operate the desktop.

In order to get results without distortion, we had a few options to run Memtest86. The first option was to set up a VM with a Ubuntu OS, because Ubuntu already includes Memtest86+ in their GRUB (boot menu). At the reboot of the VM we could select the option, but Memtest86+ is an older version of Memtest without the RAM Benchmark option. This meant we could only extract one speed value for each memory/cache hierarchies level. Therefore, we tried to set up a VM without any OS and just booting it from the newer Memtest86 version. This was possible by activating EFI in the VirtualBox settings and creating an ISO out of the created Memtest86 EFI data. Our experience with Windows computers and issuing Memtest on VMs showed only a little difference in comparison to the Memtest in the boot menu. Only one point of Memtest86 results was wrong, the newer Memtest86 offers also a detailed information about the CPU. It shows the number of the different cache elements. In our case the result was that number of L1/L2/L3 elements was exactly the same. This would mean that the cache were likely all private. But because this result were inconsistent with the hardware specification, therefore this an error likely caused by the VM environment. In the end we used the STREAM Benchmark too, just to gain more confidence about the measured results. After all results were all quite similar we concluded that they are usable for our calibration. For the detailed results see A.

The next step was to run the Java source code of [FH16] again. Both machines had more cores than 16, which was the scope of the old experiment. Hence, we attempted to compile the source code with omp4j again with a higher number in the compiler directive for parallel executed loop threads. The compilation worked but the result was

not satisfying because regardless of the value in the directive the compiled programs always did run on four cores, which we observed by using htop. Without using omp4j it did run on one core, so we could not find any reason for this behavior. The github repository did not offer any solution and the project also seemed not well maintained, because in lack of activity and bug fixing. Other than [FH16] our experiment run with Hyper-threading on, because the option could only switched off in the BIOS/EFI, where we had not access to. Observations with htop showed the expected behavior regarding the number of parallel used cores. Although the used core were not consistent the same, they were switched dynamically.

On the cache behavior side we installed perf on both machines and executed the Java programs up to 16 core in addition to perf. The program did not cause any significant overhead as we checked the measured time of a run with and with out it. After 500 runs we got a statistic about the overall cache behavior. In terms of the calibration, we did that what we mentioned in the 4.3 section and in the previous iteration. In addition, as the L1 and the L2 cache is private in our hardware the overall L1 demand is divided by the number of cores.

With that done we finished our experiment and proceeded to analyze and make use of the gathered data by plugging it into the Palladio projects, see 5.3 for the memory hierarchy in the resource diagram and 5.2. For our use-case we added the memory hierarchy which is required from the MatrixMultiplicator component. The result of that will be discussed in the next chapter.

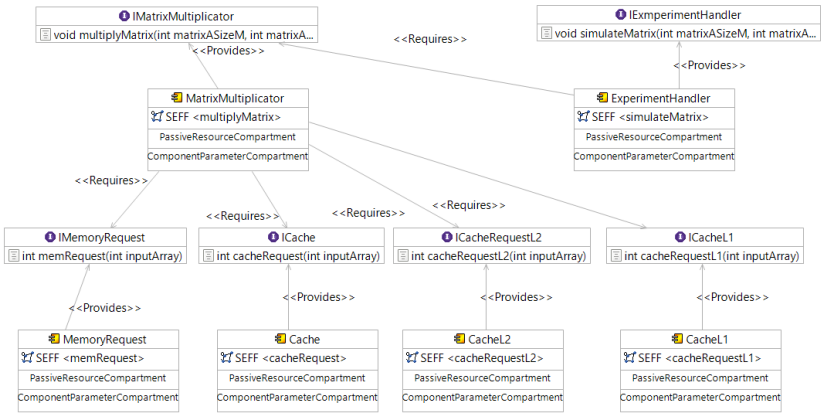

Figure 5.2.: Screenshot of the Repository Diagram of the Matrix Multiplication use-case.

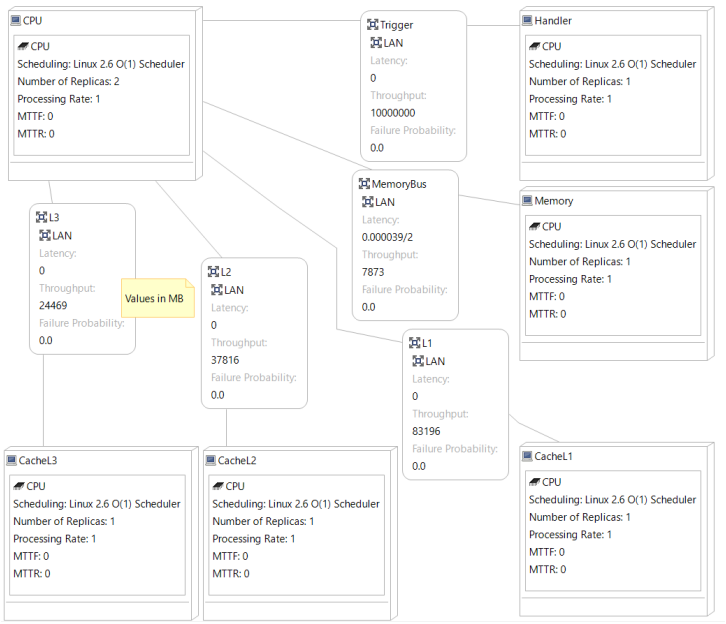

Figure 5.3.: Screenshot from Palladio Bench of the modeled memory hierarchies.

Chapter 6

Evaluation

6.1. Description of the results

In this section the results of each iteration will be briefly described.

0. Iteration prototyping:

The result of the prototyping iteration was a Palladio model which represented the RAM Benchmark of Memtest86. This prototype focuses on the actual bus demand and neglects other factors. But the prototype is able to distinguish between read and write operations as well as their different speed behavior.

1. Iteration modeling:

The plot on the next page 6.1 shows the result after the first iteration with the numbers from [FH16] ´s experiment. The result regarding the Palladio modeling was a successful combination of the so called "Memtest" prototype with the existing matrix multiplication. The calibration was problematic because we had neither access to the old server environment nor detailed information about the used hardware.

The execution time line and the old simulation time line visualize the results from A.2. Sim 2.0 is the the approach where we used the calculated numbers which were described in the calibration section, the used cache memory ratio was 70:30. In comparison to the old model approach the calibration brings an accuracy gain but with 8 and 16 cores it is now too pessimistic. The simulated memory bandwidth leads to a non linear slowdown. We also found out that our approach of bringing the memory latency into the simulation model, results in adding approximately 2000 seconds to each simulation case. We used a latency value of 21 nanoseconds, represented in the model as 21 microseconds divided by 2, because of the adjusted unit and the round trip latency. That was multiplied by the loop iterations and four memory operation of each iteration.

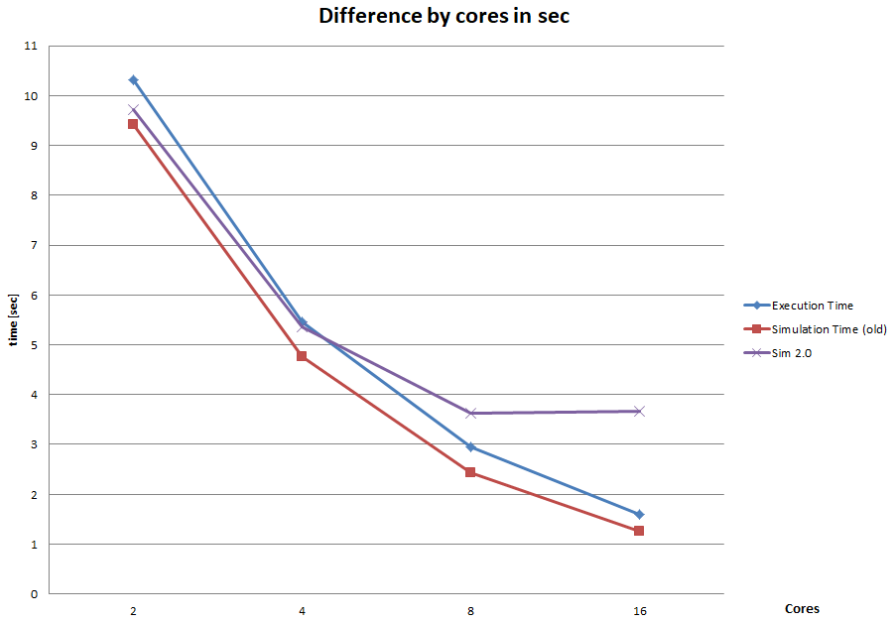

Figure 6.1.: Intermediate result on the basis of [FH16].

2. Iteration experiment:

The results of our verification experiment are visualized by the following plots 6.3 and 6.2.

They show that our predictions still differ from the reality. The theory approach with the same cache ratio is further off than the old approach from [FH16], but by measuring the cache behavior and using these values to calibrate our Palladio model we got more accurate results. The results were different on each used machine. This is the same for the cache behavior. The tendency was that with an increasing number of cores the prediction result were still too optimistic.

The 24 and the 80 have similar first level cache loads but as it is viewed in 6.4 the matrix multiplication on the 24 core machine produces more last level cache loads and misses.

Regarding the accuracy numbers, if we compare them with the numbers from the table A.2 they were roughly similar but 24 core was more accurate with the exception of the 16 core case. The 80 core machine produced more inaccurate results but there were less extreme outliers. See table A.3 and table A.4. It is worth to note that the experiment from [FH16], see in A.2, produced more uniformly accuracy results than ours. The plot A.2 which visualizes all three accuracy numbers shows exactly that.

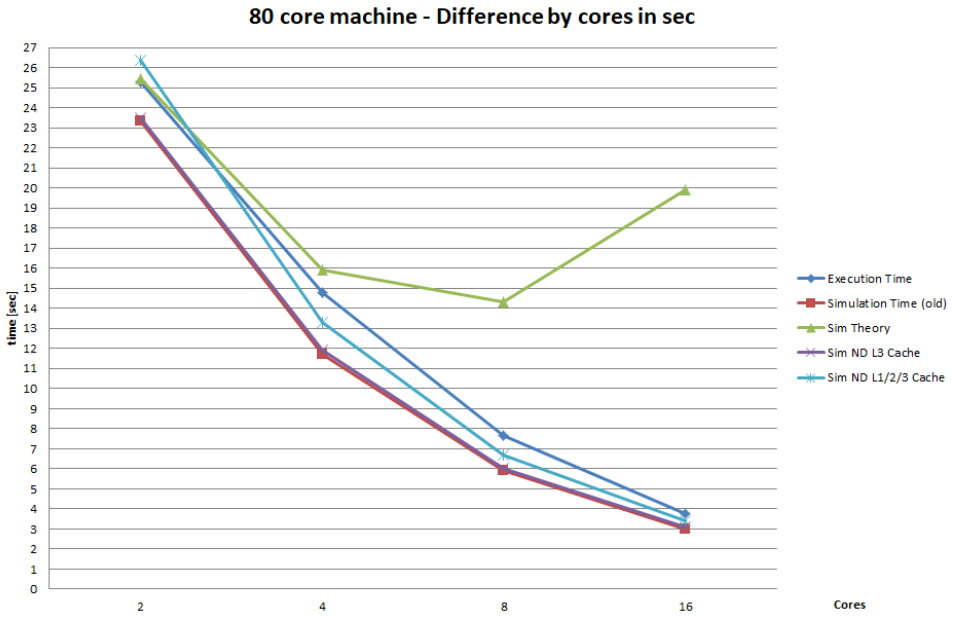

Figure 6.2.: Results for the 80 core machine.

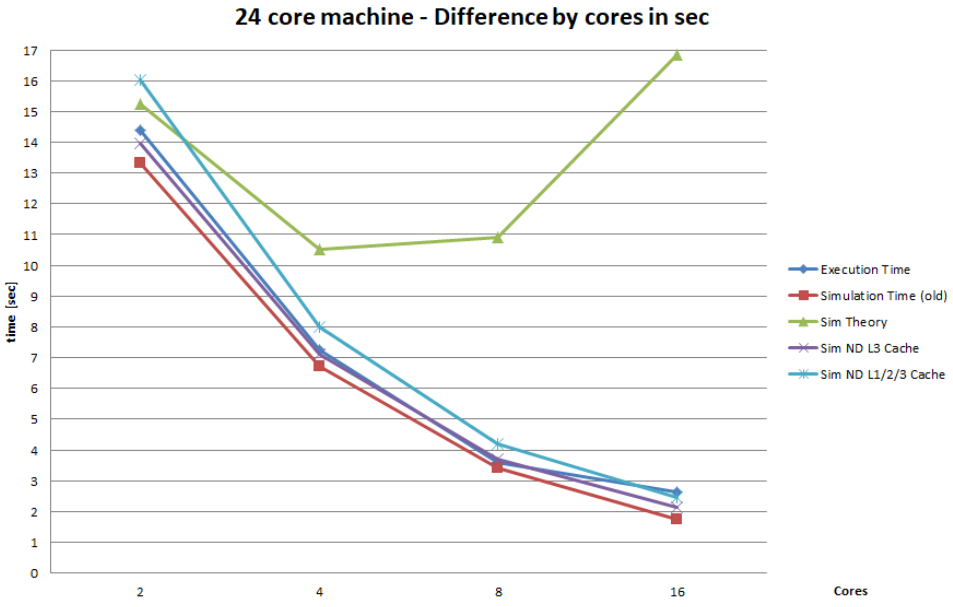

Figure 6.3.: Results for the 24 core machine.

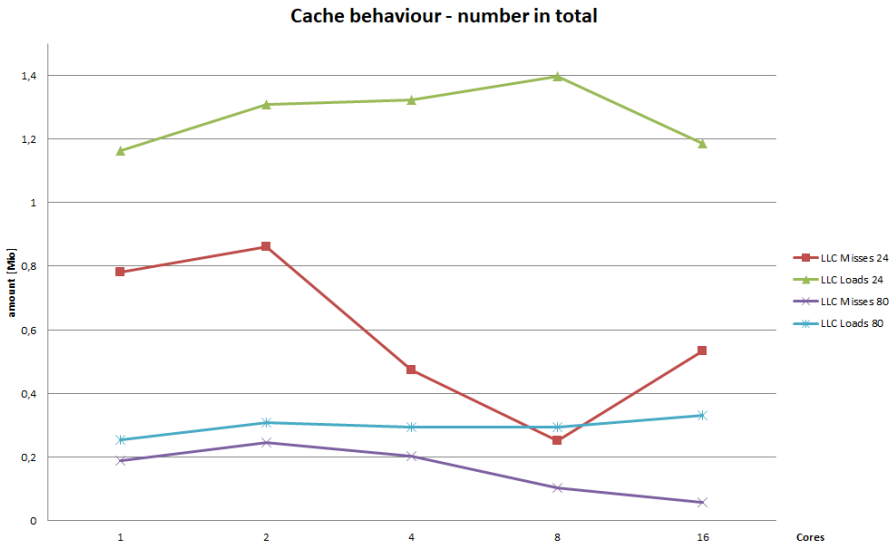

Figure 6.4.: Measured cache behavior for both experiment runs.

6.2. Discussion of the results

For the discussion of our result we will follow the postulated research questions from chapter one.

Our first research question

- RQ1) Is it possible to model multicore systems with the described approach?
  - 1.1) Is it possible to model memory bandwidth behavior of multicore systems with Palladio Bench by using network links?
  - 1.2) Is it possible to find representative for memory/cache and their size by using network links?

After it was successful to model Memtest86 with Palladio the intuitive answer is yes. But a sentiment is not enough to verify the actual state of Palladio. So let us assume a few things. First we have a main memory DRAM with the speed of 10 GB/s. Second a belonging computer system that aims to load 1 GB of data from the DRAM, before the CPU of the system is able to start a calculation. This activity requires a 1/10 of a second (Amount of data/throughput) with this values. Our Memtest86 should deliver the same result, if we plug this values into the model.

The result is exactly as expected: 0.1 sec. The case where we just add the network demand to an existing Palladio model with a certain response time. The combination leads to a new response time, which is the sum out of the "old" response time from

the model and the 0.1. As this is the conception of the memory bandwidth, Palladio is capable of modeling it. This is the same for the whole memory hierarchies, because it is possible to model each level of the hierarchy, see 5.3. We showed that in our experiment, the crucial part is the correct calibration but more on that later. The calibration itself is possible because we are able to set the throughput (speed), size of the data flow of and for each container/component, name in the Palladio terminology, or hierarchy level as equivalent.

Hence, RQ1 and 1.1/1.2 can be affirmed.

- 1.3) Are there effects we did not consider and can we model them in Palladio?
- 1.4) Are there missing parts/things in Palladio to successfully cope with multicore systems?
- 1.1.3) Did problems occur and if how we solved them?

Still we faced some flaws of Palladio which made our work kind of difficult. In theory the passive resource offer the possibility to model a limited resource, which was under consideration to get a better mapping of the shared elements of memory hierarchies. But the composition of the Exact Scheduler, the ForkAction and the Passive resources is not usable in Palladio. This enforces to prioritize the ForkAction/exact scheduler over the passive resources or vice versa. This is the reason we did not address cache contention or synchronization in our models, because we were not able to use it for our use-case. But for another use-case it could be possible to use them, with the constraint to relinquish the use of ForkActions and to be forced to the default scheduler. Therefore we would say it is less about missing parts and more about "bugs" which were hindering this approach. That comes on top to the problems [FH16] found. Although the fact that we could not use a 3000*3000*3000 loop had more consequences for our approach, because our workaround is less intuitive and more problematic. That required a course of action in order to cope with those errors was. In this case to search for a workaround, again.

So the answer to 1.3 and 1.4 is that we had problems to model cache contention effects in Palladio, because "bugs" prevented us from doing so. Therefore, bugfixes would be required to be able to do that, because we worked around those problems, in order to answer 1.1.3.

- 1.1.1) How much work does it take to model our approach in Palladio?
- 1.1.2) How many and which steps were needed to model our approach?

In terms of the usability the approach of modeling a network demand for each memory hierarchy level is less costly and error prone than the modeling of multicore programs in Palladio. The most costly and error prone part of modeling a multicore system in

Palladio is the ForkAction part, because here every core respectively replica needs its own InternalAction. In some pseudo O notation, if n is the number of cores the ForkAction modeling requires O(n) modeling steps as the modeling of the memory hierarchy is O(1), so a constant factor relating to an increasing number of cores. That is the answer for 1.1.1 and 1.1.2.

Looking at our hypothesis H1

- H1) We are able to model multicore systems with the network links.

we can assume that this holds true. Now we are able to reject the null hypothesis: That it is not possible to model a memory bandwidth behavior in Palladio.

Coming to the second research question.

- RQ2) Is the prediction more accurate?

The plots from the result description visualize the different stages of accuracy gains with our approach. Referring to the numbers, the question if our approach enables more accurate results can be affirmed, too. This is coherent as we remarked above, because the combination of a memory bandwidth simulation with an existing project leaded to a sum of both response times. This holds true for the most cases but with some reservations, because by using a network demand within a ForkAction we noticed sometimes the behavior of a one tenth second faster result than without it. Therefore, it is more interesting to look at the actual gain. Our hypothesis H2 was:

- H2) The prediction is more accurate and reaches the 90 percent accuracy.

This is true in some way but not for every case. The improvements were not fully consistent because in the plot the result line cut at least once the measured execution time line. That means automatically that the predictions were before the cut to optimistic and afterwards to pessimistic or the other way round.

The biggest issue is, in the accurate cases we rely on factors that do not increase with the number of cores. Due to this fact for more cores than 16 we would sooner or later see a bigger gap. In addition our trial to bring in a factor that increases with the number of cores failed. The initial trial by simulating the compressed distribution of the memory bus traffic lead to a big overestimation. With the measured cache behavior we still found no factor which followed the non linear factor closely. That did not change regardless if we modeled the full memory hierarchy or only a part of it.

In order to answer the remaining sub research questions one by one.

- 2.1) How accurate are the results in comparison to the performance in reality?

The accuracy differs from approach to approach and machine to machine. In the appendix the full results were listed, the average prediction error for the approach to model the full memory hierarchies was about 9% for the 80 core machine and about 11.5% for the 24 core machine. The simulation of just the L2/L3 plus the DRAM resulted in 16.5% for the 80 core case and 5.5% as average prediction error for the 24 core machine. Altogether the results were approximately 10% off the measured performance in reality. The simulations that only simulated the shared cache and memory had an prediction error of 16.25% (80 core machine) and 5.5% (24 core machine). It is interesting to note that the simulation without the level one cache helped to get more accurate results for the 80 core machine but on the other hand was really bad for the 24 cores machine case. In addition the simulation of the L2 cache did not change the results significantly.

- 2.2) How accurate are the results in comparison to the "default" or single core Palladio simulations?

In order to judge about that we look at the average prediction error of the default modeling, which were 13% for the 24 core machine and 18% for the 80 core machine. In comparison to our average prediction error we have in the average a gain of 5%. But these gains were made especially on the results which the default modeling was far off. The rules of thumb is, the more core and the more cache misses our approach has his strength against the default. But for the two core case the gains were marginal.

- 2.3) How accurate are the results in comparison to the performance of other multicore simulation approaches?

Two from three of the performance models we described more closely in this thesis stated the accuracy of their model by a number. Those were at 1.57% respectively 4.5% for the average prediction error. That means they were better than our 10% average prediction error but as stated in the related work section, the first number relies on limited testing.

6.3. Threats to validity

There are several threads to the validity. One threat is that we are model in our approach the memory bandwidth in the form of a network demand that is contributing to the overall response time. This is shown by the difference between the calculated theoretical and measured values. The memory bandwidth is becoming a factor, if the CPU is idle because it is waiting for data to proceed. But our model is to model the time which is needed to transfer the data. Those two are not the same, in the best case we are just missing the overhead and the rest is quite similar.

This brings us to the next threat: Our assumptions. We assume that the memory bus is a bottleneck. But we have no idea to what amount the CPU waits in reality because of the memory bandwidth. This is the biggest threat, because we do not know for sure if and how the memory bandwidth is a bottleneck for the matrix multiplication use-case.

Or more general we are lacking in a foundation knowledge about the exact memory behavior. In order to prove this it would be necessary to observe the degree of utilization of the memory bus and the whole "memory hierarchy chain." In addition we assume our calibration is accurate enough to estimate the speed of the memory hierarchies but this is still a approximation. The whole experiment with theoretical numbers failed and proved to be very inaccurate. We assume the reason for these results were the missing cache numbers, because it got better with them. In addition the measured cache numbers were consistent to our calculations, because A.3 also showed 3 load operation per loop iteration. That was the same as what we assumed in the 4.3 section. But we had no measured numbers at all for the write operations and therefore skipped them, as well as we only issued the latency divided by two per overall network demand.

Hence, those numbers were still inadequately to represent the "memory bus" or the memory transfer from the CPU to the memory hierarchy. It would be better if we could exactly follow the data flow through the "memory hierarchy chain" and have access to all entire picture. The entire picture includes also the hardware structure, our models respected that each core had its own L1/L2 cache, therefore we divided the network demand by the number of these caches. But this gets more complicated for the L3 cache, because they were shared but each CPU element, the 24 core machine had four and the 80 core machine eight. We were not able to observe which of them were used in which use-case. Indeed htop allows in Linux to observe which core is used and to what percentage but it is not clear how they were mapped. Especially because hyper threading was not switched off. The perf results also showed the number of utilized CPU´s and we got the expected value, but makes no assertion about the mapping. Although the points "cpu-migrations" and "context-switches" indicate something in this direction, nevertheless it is not enough to change this.

In an ideal scenario in the four/eight core matrix multiplication scenario each thread had its own L3 cache. From 8/16 threads both threads had to share the L3 caches. But this behavior was a black-box for us. Therefore, we ignored the number of L3 caches in our models as well as the write operations, the latency and potential contention, synchronization effects in the shared resources. The renouncement of them is no statement that those factors were unimportant and do not contribute to the non-linear speedup. Quite the contrary but we could just not include a factor which increases the response time per core. In addition our experiment showed such a behavior exists and the 16 thread case on the 24 core machine is the best example. It it is worth to note that this results could be replicated and was not a result of a non-recurring event.

Chapter 7

Conclusion

In conclusion this thesis showed that it is possible to model the memory bandwidth with existing elements in Palladio. It is not straight forward and the user has to deal with several issues and make use of workarounds to cope with them. But it is a reasonable approach to improve the precision of multicore predictions.

The results or the accuracy gain is significant but it is clear that we did not solve the problem of Palladio´s inaccuracy completely by modeling the memory bandwidth. In particular we did not integrate a factor into our models, which increased with the number of used cores. The experiment brought diverse results, with in general more accurate predictions than the standard modeling, introduced by [FH16]. But the prediction error of approximately 10% leaves space for further improvements.

7.1. Discussion and benefits

This thesis had the target to examine the capabilities of Palladio in terms of mapping complex CPU architecture to Palladio hardware models and evaluate the accuracy of the performance predictions. The result is that Palladio is able to model some parts of these complex CPU architectures to improve the accuracy. But in order to solve the whole problem this was probably the wrong way round. It was stated before that we lacked in knowledge of the entire memory hierarchy chain behavior.

We examined the cache behavior, see 6.4. The result showed that the 24 core machine has more LLC loads/misses than the 80 core because of the fact that more core/caches allow a more ideal distribution of cache loads. In addition the increasing number of LLC cache misses explains why the 16 thread multiplication had a relative weak speedup. Likely due to the fact that 16 threads shared four L3 cache elements. But the cache behavior in general remained some sort of a black box for us, because measuring is likely

not a desired way to calibrate the performance predictions. In my opinion it would have been easier to first have the entire picture of the memory behavior or which factors are the reason for the non-linear speedup of multicore programs. After this it is easier to try to model them in Palladio and to judge if it is possible to map those factors to the Palladio hardware models.

Anyway the benefit of these findings is that we could show the capabilities of Palladio, with the limitation that there is still room for improvement. Since the predictions were not accurate enough, even with a close calibration. This was shown by our experiments, which included a repetition of [FH16]. But since we had problems with omp4j we could not extend the experiment to more cores. Furthermore compatibility problems with passive resources prevented us from using the in our experiment, too. For this reasons it seems to be recommended to change the framework and to fix the specific bug in Palladio.

7.2. Future work and outlook

The entry point for future work is that this thesis did not find the factor or the factors which increase with the core size. This would be the requirement to get (more) accurate predictions, whether it is the cache contention/synchronization or something different. As said it would be easier to improve Palladio´s multicore predictions by knowing what is the exact reason for the inaccuracy.

In the chapter before we stated that the related work did focus in the most cases on the topic caches and cache contention. The work of [VE11], [XCDM10] and in particular [EHW09] place emphasis on those issues.

Therefore, it seems reasonable to investigate this next. Another factor is the synchronization of the shared memory resources [BA09]. As stated before the problems with the passive resources make it challenging to bring these effects into Palladio by now. Furthermore, the flaws of Palladio regarding bugs/errors or a cumbersome modeling of multicore systems do not help. Bug fixes and the possibility of extending the AT, to save modeling effort could be helpful in terms of usability but less to get better results. This is not the only part, [BKR09] stated the possibility to bring the cache behavior into Palladio, for example as special case of the de-/marshalling of the network demand. This seems to be an option to implement the cache behavior in a more efficient way.

Regarding the experiment, it would be interesting to see how the matrix multiplication speed up is developing with more than 16 cores. These observations could help to get a better understanding of the hardware behavior. A close/detailed observation of the memory/cache behavior could be useful in this concern. Hence, further experiments

with the matrix multiplication use case are recommended but as stated probably with another framework and with more measurements. They should be done in order to get the entire picture of the memory hierarchies behavior.

To sum this up:

Aspects covered by this thesis:

- Palladio is capable of modeling the memory bandwidth for the memory hierarchies.
- The modeled memory bandwidth improves the accuracy.
- The caching behavior of the matrix multiplication use case was examined and measured.
- Caching(loads/misses) can be added to the Palladio model.
- The results of [FH16] ´s experiment were confirmed.

Aspects that were not (fully) covered by this thesis and should be done in the future:

- Investigate the "full" behavior of the memory hierarchies, which is supposed to be the predominate factor for the non linear slowdown.
- Gather detailed information about the cache/memory behavior (more than hit/miss, e.g. time data needs to get from the memory to the CPU and contention/synchronization effects).
- Adding synchronization or cache contention to the Palladio model.
- Run the matrix multiplication experiment with more than 16 cores and an adapted setting.

Appendix A

Appendix

In this chapter the data of our experiment results will be presented.

Calculations

(A.1)

$$\begin{aligned}
&\text{max. Memory Bandwidth} = 800\,\text{MHz} * 2 * 64\,\text{Bit} * 2 \quad | \text{ (DDR = double data rate)} \\
&\text{max. Memory Bandwidth} = 20480000000\,\text{Bit/s} \quad | \text{ (devide by 8)} \\
&\text{max. Memory Bandwidth} = 25600000000\,\text{Byte/s} \\
&\text{max. Memory Bandwidth} = 25.6\,\text{GB/s}
\end{aligned}$$

(A.2)

$$\begin{aligned}
&\text{Calibration} = 18.64\,\text{sec} / (3000 * 3000 * 3000) \\
&\text{Calibration} = 0.69\,\text{ns}
\end{aligned}$$

(A.3)

$$\begin{aligned}
&\text{L1 cache loads per loop iteration} = 81330354213\,\text{loads} / (3000 * 3000 * 3000) \\
&\text{L1 cache loads per loop iteration} = 3.02\,\text{loads/iteration}
\end{aligned}$$

Calibration results

For the 24 core machine on average and approved by the STREAM benchmark:

- L1 32K: 81.196 GB/s
- L2 256K: 37.816 GB/s
- L3 15M: 24.469 GB/s
- Memory: 7.873 GB/s

For the 80 core machine on average and approved by the STREAM benchmark:

- L1 32K: 59.858 GB/s
- L2 256K: 31.504 GB/s
- L3 30M: 21.966 GB/s
- Memory: 4.776 GB/s

The following table A.1 shows the issued amount of traffic for the the "Sim Theory" approach.

Experiment results

Table A.1.: Theoretical memory bandwidth for the experiment of [FH16].

Threads	Linear execution time (s)	Memory demand per second (GB/s)
1	18.64	23.5
2	9.32	46.35
4	4.66	92.7
8	2.33	185.4
16	1.165	370.8

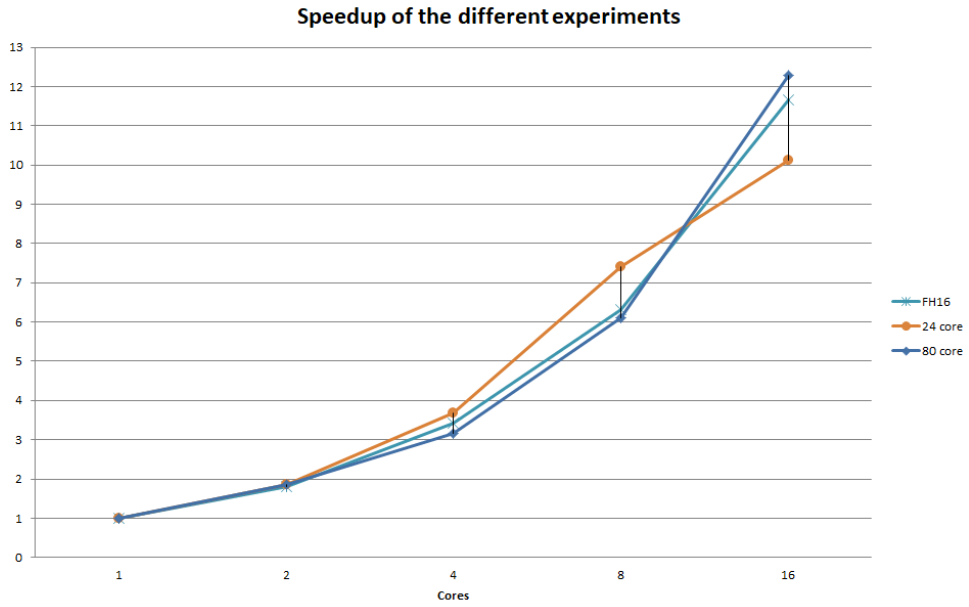

Figure A.1.: Speedup of the experiments in comparison.

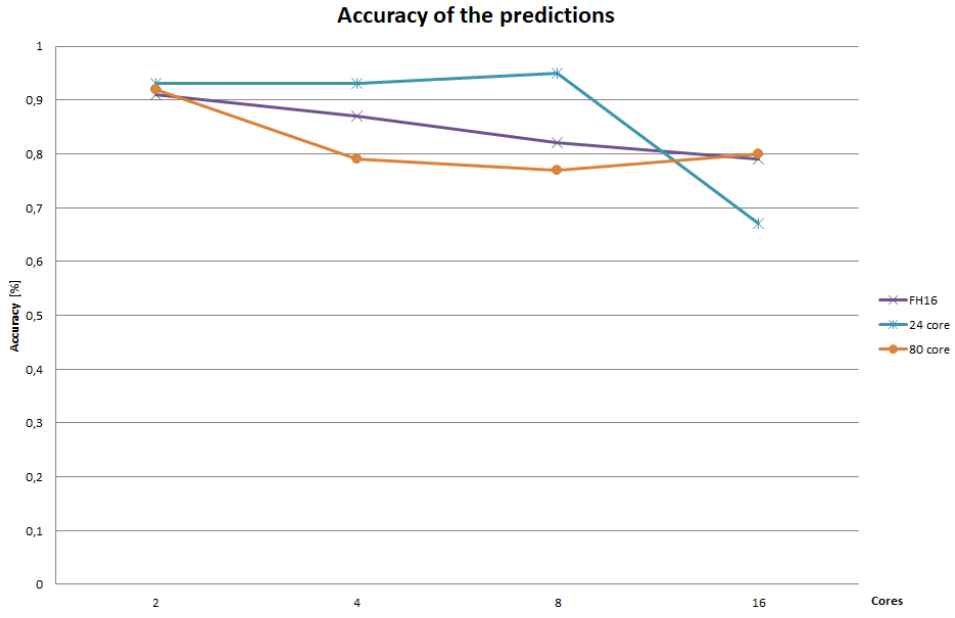

Figure A.2.: Accuracy numbers of the repetition of the [FH16] experiment in comparison.

Table A.2.: Experiment results and accuracy of [FH16].

Threads	Mean execution (s)	Mean prediction (s)	Accuracy	Speedup
1	18.64	18.63	0.99	1
2	10.31	9.41	0.91	1.81
4	5.45	4.76	0.87	3.42
8	2.95	2.43	0.82	6.32
16	1.60	1.26	0.79	11.66

Table A.3.: Repetition of the [FH16] experiment on the 24 core machine.

Threads	Mean execution (s)	Mean prediction (s)	Accuracy	Speedup
1	26.623	26.460	0.99	1
2	14.403	13.33	0.93	1.85
4	7.241	6.715	0.93	3.68
8	3.593	3.407	0.95	7.41
16	2.632	1.753	0.67	10.12

Table A.4.: Repetition of the [FH16] experiment on the 80 core machine.

Threads	Mean execution (s)	Mean prediction (s)	Accuracy	Speedup
1	46.667	46.440	1	1
2	25.275	23.320	0.92	1.85
4	14.775	11.710	0.79	3.16
8	7.645	5.905	0.77	6.10
16	3.766	3.002	0.8	12.29

Table A.5.: Results of the different modelings based on the 24 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L1/L2/L3 (s)	Accuracy	Prediction L2/L3 (s)	Accuracy
2	16.025	1.11	14.067	0.98
4	8.015	1.11	7.015	0.97
8	4.207	1.17	3.707	1.03
16	2.453	0.93	2.253	0.86

Table A.6.: Results of the different modelings based on the 80 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L1/L2/L3 (s)	Accuracy	Prediction L2/L3 (s)	Accuracy
2	26.35	1.04	23.641	0.94
4	13.31	0.9	11.91	0.81
8	6.705	0.88	6.005	0.79
16	3.402	0.9	3.102	0.82

Table A.7.: Results of the different modelings based on the 24 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L1/L2/L3 (s)	Accuracy	Prediction L2/L3 (s)	Accuracy
2	16.025	1.11	14.067	0.98
4	8.015	1.11	7.015	0.97
8	4.207	1.17	3.707	1.03
16	2.453	0.93	2.253	0.86

Table A.8.: Results of the different modelings based on the 80 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L1/L2/L3 (s)	Accuracy	Prediction L2/L3 (s)	Accuracy
2	26.35	1.04	23.641	0.94
4	13.31	0.9	11.91	0.81
8	6.705	0.88	6.005	0.79
16	3.402	0.9	3.102	0.82

Table A.9.: Results of the different modelings based on the 24 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L3 (s)	Accuracy	Prediction L1/L3 (s)	Accuracy
2	13.967	0.97	14.067	0.98
4	7.115	0.98	7.015	0.97
8	3.707	1.03	3.707	1.03
16	2.253	0.086	2.253	0.86

Table A.10.: Results of the different modelings based on the 80 core machine experiment. The accuracy is in comparison to the corresponding execution time above.

Threads	Prediction L3 (s)	Accuracy	Prediction L1/L3 (s)	Accuracy
2	23.527	0.93	26.35	1.04
4	11.91	0.81	13.21	0.89
8	6.005	0.79	6.705	0.88
16	3.102	0.82	3.402	0.9

Table A.11.: Measured Cache behavior on the 24 core machine.

Threads	Mean L1 loads	Mean L1 misses	Mean LLC loads	Mean LCC misses
1	81 330 354 213	1 762 383 262	1 162 751 113	782 521 154
2	81 763 678 942	1 826 813 810	1 309 399 619	860 986 776
4	81 721 379 308	1 855 617 335	1 321 395 716	472 791 373
8	81 767 359 484	1 875 524 408	1 395 814 042	253 153 313
16	83 178 175 048	2 493 550 882	1 185 124 499	533 684 834

Table A.12.: Measured Cache behavior on the 80 core machine.

Threads	Mean L1 loads	Mean L1 misses	Mean LLC loads	Mean LCC misses
1	81 437 203 252	1 759 762 679	253 916 003	190 525 904
2	81 450 865 750	1 804 534 522	309 306 840	247 616 799
4	81 469 553 810	1 827 521 651	293 814 941	203 089 341
8	81 454 945 487	1 839 216 550	294 379 396	105 065 108
16	81 434 130 183	1 860 642 925	331 221 033	57 772 173

Appendix A

Bibliography

[BA09] B. B. Brandenburg, J. H. Anderson. "Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems." In: *2009 21st Euromicro Conference on Real-Time Systems*. July 2009, pp. 184–193. DOI: 10.1109/ECRTS.2009.14 (cit. on p. 52).

[Bec08] S. Becker. "Coupled Model Transformations for QoS Enabled Component-Based Software Design." PhD thesis. University of Oldenburg, Germany, 2008. DOI: 10.5445/KSP/1000009095 (cit. on pp. 14, 28, 29).

[BKR09] S. Becker, H. Koziolek, R. Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066 (cit. on pp. 1, 15, 25, 27, 29, 30, 52).

[BXG12] J. Bui, C. Xu, S. Gurumurthi. *Understanding Performance Issues on both Single Core and Multi-core Architecture*. 2012. URL: https://pdfs.semanticscholar.org/b5f3/92abf1d2c937197eb86f601344c842c4f0fe.pdf (cit. on p. 8).

[CGKS05] D. Chandra, F. Guo, S. Kim, Y. Solihin. "Predicting inter-thread cache contention on a chip multi-processor architecture." In: *11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pp. 340–351. DOI: 10.1109/HPCA.2005.27 (cit. on p. 23).

[DK00] C. Ding, K. Kennedy. "The Memory Bandwidth Bottleneck and Its Amelioration by a Compiler." In: *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*. IPDPS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 181–189. ISBN: 0-7695-0574-0. URL: http://dl.acm.org/citation.cfm?id=846234.849354 (cit. on p. 8).

[EHW09] J. Eitzinger, G. Hager, G. Wellein. “Multi-core architectures: Complexities of performance prediction and the impact of cache topology.” In: (Oct. 2009). URL: https://arxiv.org/abs/0910.4865 (cit. on p. 52).

[FH16] M. Frank, 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://www.researchgate.net/publication/309858146_Performance_Prediction_for_Multicore_Environments-_An_Experiment_Report (cit. on pp. 4–6, 19, 25, 32, 33, 37–40, 43, 44, 47, 51–53, 56–58).

[FHB18] M. Frank, M. Hilbrich, S. Becker. “Modeling Language Enhancement to enable Parallel Constructs in Software Performance Models.” In: *ICPE ’18: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. Berlin, Germany: ACM, 2018. ISBN: 978-1-4503-5095-2. URL: http://doi.acm.org/10.1145/3185768.3185773 (cit. on pp. 16, 17).

[FHLB17] M. Frank, M. Hilbrich, S. Lehrig, S. Becker. “Parallelization, Modeling, and Performance Prediction in the Multi-/Many Core Area: A Systematic Literature Review.” In: *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*. Nov. 2017, pp. 48–55. DOI: 10.1109/SC2.2017.15 (cit. on pp. 1, 7).

[FKB18] M. Frank, F. Klinaku, S. Becker. “Challenges in Multicore Performance Predictions.” In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. Berlin, Germany: ACM, 2018, pp. 47–48. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3185773. URL: http://doi.acm.org/10.1145/3185768.3185773 (cit. on pp. 2, 3).

[Gre19] B. Gregg. *perf Examples*. Retrieved in April 20th. 2019. URL: http://www.brendangregg.com/perf.html (cit. on p. 13).

[Hap09] J. Happe. “Predicting software performance in symmetric multi-core and multiprocessor Environments.” PhD thesis. 2009. 291 pp. ISBN: 978-3-86644-381-5. DOI: 10.5445/KSP/1000011806 (cit. on pp. 26, 27).

[HM08] M. D. Hill, M. R. Marty. “Amdahl’s Law in the Multicore Era.” In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 0018-9162. DOI: 10.1109/MC.2008.209. URL: http://research.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf (cit. on p. 8).

[HM17] M. D. Hill, M. R. Marty. “Retrospective on Amdahl’s Law in the Multicore Era.” In: *Computer* (July 2017), pp. 40–42. URL: http://research.cs.wisc.edu/multifacet/papers/ieeecomputer17_amdahl_multicore_retro.pdf (cit. on p. 8).

[Leh18] S. M. Lehrig. “Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge.” PhD thesis. Karlsruher Institut für Technologie (KIT), 2018. 514 pp. ISBN: 978-3-7315-0756-7. DOI: 10.5445/KSP/1000079766 (cit. on p. 17).

[McC07] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: http://www.cs.virginia.edu/stream/ (cit. on pp. 8, 11).

[McC95] J. McCalpin. “Memory bandwidth and machine balance in high performance computers.” In: *IEEE Technical Committee on Computer Architecture Newsletter* (Dec. 1995), pp. 19–25. URL: https://www.researchgate.net/publication/51992086_Memory_bandwidth_and_machine_balance_in_high_performance_computers (cit. on p. 11).

[MOO75] G. E. Moore. In: *Proceedings of the International Electron Devices Meeting*. IEEE, 1975, pp. 11–13. URL: http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf (cit. on p. 7).

[RBH+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, K. Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016. URL: https://dl.acm.org/citation.cfm?id=3036121 (cit. on p. 1).

[RBK+07] R. H. Reussner, S. Becker, H. Koziolek, J. Happe, M. Kuperberg, K. Krogmann. *The Palladio Component Model*. Interner Bericht 2007-21. October 2007. Universität Karlsruhe (TH), 2007. URL: http://sdqweb.ipd.uka.de/publications/pdfs/reussner2007a.pdf (cit. on p. 1).

[RBK+11] R. H. Reussner, S. Becker, H. Koziolek, J. Happe, M. Kuperberg, K. Krogmann, E. Berger, A. Koziolek, M. Hauk. *The Palladio Component Model*. Tech. rep. Universität Karlsruhe (TH), 2011. DOI: 10.5445/IR/1000022503 (cit. on pp. 27, 29, 30).

[VE11] K. Van Craeynest, L. Eeckhout. “The Multi-Program Performance Model: Debunking current practice in multi-core simulation.” In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2011, pp. 26–37. DOI: 10.1109/IISWC.2011.6114194 (cit. on pp. 22, 52).

[WWP09] S. Williams, A. Waterman, D. Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009. DOI: 10.1145/1498765.1498785 (cit. on pp. 12, 19, 20).

[XCDM10] C. Xu, X. Chen, R. P. Dick, Z. M. Mao. “Cache Contention and Application Performance Prediction for Multi-Core Systems.” In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. Mar. 2010, pp. 76–86. DOI: 10.1109/ISPASS.2010.5452065 (cit. on pp. 23, 24, 52).

[YP10] C. Yu, P. Petrov. “Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms.” In: *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*. 2010, pp. 132–137. DOI: 10.1145/1837274.1837309. URL: http://doi.acm.org/10.1145/1837274.1837309 (cit. on p. 8).

All links were last followed on April 28, 2019.

Declaration

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

---

place, date, signature