Performance Improvements Using Dynamic Performance Stubs

PhD Thesis

Peter Trapp

Software Technology Research Laboratory
De Montfort University

This thesis is submitted in partial fulfillment of the requirements for the Doctor of Philosophy.

July 2011

To my family.

Abstract

If you think performance is expensive, try using a paralyzed system.

...

This thesis proposes a new methodology to extend the software performance engineering process. Common performance measurement and tuning principles mainly target to improve the software function itself. Hereby, the application source code is studied and improved independently of the overall system performance behavior. Moreover, the optimization of the software function has to be done without an estimation of the expected optimization gain. This often leads to an under- or over-optimization, and hence, does not utilize the system sufficiently.

The proposed performance improvement methodology and framework, called *dynamic performance stubs*, improves the before mentioned insufficiencies by evaluating the overall system performance improvement. This is achieved by simulating the performance behavior of the original software functionality depending on an adjustable optimization level prior to the real optimization. So, it enables the software performance analyst to determine the systems' overall performance behavior considering possible outcomes of different improvement approaches. Moreover, by using the *dynamic performance stubs* methodology, a cost-benefit analyses of different optimizations regarding the performance behavior can be done.

The approach of the *dynamic performance stubs* is to replace the software bottleneck by a stub. This stub combines the simulation of the software functionality with the possibility to adjust the performance behavior depending on one or more different performance aspects of the replaced software function. A general methodology for using *dynamic performance stubs* as well as several methodologies for simulating different performance aspects are discussed. Finally, several case studies to show the application and usability of the *dynamic performance stubs* approach are presented.

Declaration

I hereby declare that this thesis is my own work undertaken by me between July 2007 and July 2011 for the degree of Doctor of Philosophy. It is submitted at the Software Technology Research Laboratory (STRL), De Montfort University, United Kingdom. I have not used any sources or aids other than those stated and marked verbatim and indirect quotations as such.

Publications

- Dynamic Performance Stubs:
  - Journals:
    1. Peter Trapp, Markus Meyer, Christian Facchi, Helge Janicke, and François Siewe. *Building CPU Stubs to Optimize CPU Bound Systems: An Application of Dynamic Performance Stubs.* International Journal on Advances in Software, 4(1&2), 2011. (accepted paper)
  - Conferences and Workshops:
    1. Markus Meyer, Helge Janicke, Peter Trapp, Christian Facchi and Marcel Busch. *Performance Simulation of a System's Parallelization.* In ICSEA '11: Proceedings of the International Conference on Software Engineering Advances. 2011. (accepted paper)
    2. Peter Trapp, Markus Meyer and Christian Facchi. *Dynamic Performance Stubs to Simulate the Main Memory Behavior of Applications.* In SPECTS '11: Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems. IEEE Communications Society, 2011.
    3. Peter Trapp, Markus Meyer and Christian Facchi. *Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs.* In ICSEA '10: Proceedings of the International Conference on Software Engineering Advances. IEEE Computer Society, 2010. (Best Paper Award)
    4. Peter Trapp and Christian Facchi. *Main Memory Stubs to Simulate Heap and Stack Memory Behavior.* In Computer Measurement Group 2010: International Conference Proceedings. Computer Measurement Group, Orlando (FL), 2010.

5. Peter Trapp, Christian Facchi and Markus Meyer. *Echtzeitverhalten durch die Verwendung von CPU Stubs: Eine Erweiterung von Dynamic Performance Stubs.* In Workshop "Echtzeit 2009 - Software-intensive Verteilte Echtzeitsysteme". Springer-Verlag, Boppart (Germany), 2009.
6. Peter Trapp, Christian Facchi and Sebastian Bittl. *The Concept of Memory Stubs as a Specialization of Dynamic Performance Stubs to Simulate Memory Access Behavior.* In Computer Measurement Group 2009: International Conference Proceedings. Computer Measurement Group, Dallas (TX), 2009.
7. Peter Trapp and Christian Facchi. *How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs.* In Computer Measurement Group 2008: International Conference Proceedings, pages 343-353. Computer Measurement Group, Las Vegas (NV), 2008.

- Technical Reports:

1. Peter Trapp, Markus Meyer and Christian Facchi. *How to Correctly Simulate Memory Allocation Behavior of Applications by Calibrating Main Memory Stubs.* Technical Report 20, Ingolstadt University of Applied Sciences, May 2011.
2. Peter Trapp and Christian Facchi. *Performance Improvement Using Dynamic Performance Stubs.* Technical Report 14, Ingolstadt University of Applied Sciences, August 2007.

- Software Performance Engineering:

  - Workshops:

1. Christian Facchi, Peter Trapp and Jochen Wessel. *Metrics and SCRUM in Real Life – Enemies or Friends?* In SMEF 2011 (Software Measurement Europe Forum). Rome, Italy.
2. Christian Facchi, Peter Trapp and Jochen Wessel. *Enhancing Continuous Integration by Metrics and Performance Criteria in a SCRUM Based Process - Metrics and SCRUM in an Industrial Environment: A Contradiction?* In EPIC 2010 (Workshop on Leveraging Empirical Research Results for Software Business Success). Bolzano, Italy.

Acknowledgements

First of all, I like to thank Christian Facchi for his continuing support, advises and contributions during my studies and in all matters.

I would like to thank my supervisors François Siewe and Hussein Zedan for their guidance and critical analysis for the completion of this thesis. Many thanks go to Lynn Ryan and Lindsey Trent for their ongoing support.

Special thanks go to my friends and colleagues Helge Janicke, Markus Meyer, Sebastian Röglinger and Alexander Ost. They contributed in so many ways. Thank you for that.

My gratitude is to my family. This thesis would not have been possible without them. I love you.

I like to thank our industrial partner for their ongoing and thoroughly support for this thesis. They provided an industrial environment for validating our approaches. Especially, I like to thank: Rudi Bauer, Oliver Korpilla, John Mackenzie, Karl Mattern, Jörg Monschau, Florian Oefelein, Marco Seelmann, Helmut Voggenauer and Jochen Wessel.

Thank you!

Contents

List of Figures **vii**

List of Tables **viii**

List of Listings **ix**

1 Introduction **1**

2 Literature Review on Performance Engineering **6**

2.1 Introduction to Performance Engineering 7
2.2 History of Performance Engineering 7
2.3 Areas of Software Performance Engineering 8
2.3.1 Performance Measurements 9
2.3.2 Performance Modeling 9
2.4 Performance Measurements Studies 10
2.4.1 Performance Target Specification 11
2.4.2 Traffic Model 11
2.4.3 Performance Test Environment 12
2.4.4 Measurement Study 13
2.5 Performance Measurement Tools 16
2.5.1 Mode of Operation 16
2.5.2 Characteristics 19
2.5.3 Classification 19

3 Literature Review in the Area of Dynamic Performance Stubs **22**

3.1 Dynamic Performance Stubs 23
3.2 CPU Stubs 23
3.2.1 Related Work 23
3.2.2 Basic Literature 24

3.3 Main Memory Stubs . . . 26
3.3.1 Related Work . . . 26
3.3.2 Basic Literature . . . 27
3.4 Data Cache Memory Stubs . . . 30
3.4.1 Related Work . . . 30
3.4.2 Basic Literature . . . 31
3.5 Simulated Software Functionality . . . 42
3.5.1 Related Work . . . 43
3.5.2 Basic Literature . . . 43

4 Dynamic Performance Stubs **46**
4.1 Basic Design Decisions . . . 47
4.2 Concept . . . 47
4.3 Framework . . . 48
4.4 Performance Simulation Functions . . . 49
4.4.1 CPU PSF . . . 50
4.4.2 Memory PSF . . . 51
4.4.3 I/O PSF . . . 51
4.4.4 Network PSF . . . 52
4.4.5 Calibration Functions . . . 52
4.5 Simulated Software Functionality . . . 53
4.6 Definition of a General Methodology . . . 54
4.7 Extensions to the Methodology . . . 56
4.7.1 Mixture of PSF . . . 56
4.7.2 Full and Partial Stubs . . . 57
4.7.3 Idealized Measurements . . . 57
4.7.4 Load and Stress Tests . . . 57
4.7.5 Hidden Bottlenecks Detection by Zero Bound CUS . . . 57
4.7.6 System Bounds . . . 58
4.7.7 Global vs. Local Stubs . . . 58
4.8 Advantages . . . 59
4.9 Restrictions . . . 60
4.10 Summary . . . 60

5 CPU Stubs **62**
5.1 Requirements . . . 63

5.2 Realization of the CPU Performance Simulation Functions 63
5.3 Calibration Functions 65
5.4 Methodology 67
5.5 Case Study 73
5.5.1 Evaluation Environment 73
5.5.2 Original Function 74
5.5.3 CPU Stubs 74
5.6 Summary 76

6 Main Memory Stubs 78
6.1 Requirements 79
6.2 Realization of the Main Memory Performance Simulation Functions . 80
6.2.1 Simulation Data File 82
6.2.2 Algorithm 83
6.3 Calibration Functions 85
6.3.1 Measurement Tools 85
6.3.2 Overhead Determination 85
6.4 Simulation Data File Generation 90
6.4.1 Requirements 90
6.4.2 Algorithm 91
6.4.3 Simulation Data Point Calculation 93
6.5 Methodology 99
6.6 Case Study 101
6.6.1 Evaluation Environment 101
6.6.2 Original Function 102
6.6.3 Main Memory Stubs 102
6.7 Summary 109

7 Data Cache Memory Stubs 111
7.1 Requirements 112
7.2 Realization of the Data Cache Memory Performance Simulation Functions 113
7.2.1 Architecture Description File 114
7.2.2 Simulation Data File 115
7.2.3 Algorithm 116
7.2.4 Discussion 122

7.2.5 Replacement Policies 124
7.3 Calibration Functions 128
7.4 Methodology 130
7.5 Case Study 133
7.5.1 Evaluation Environment 133
7.5.2 Original Function 134
7.5.3 Data Cache Memory Stubs 135
7.6 Summary 138

8 Simulated Software Functionality 139
8.1 Requirements 140
8.2 Methodology 142
8.3 Realization 143
8.3.1 Generate Header File (Serialization Specification) 144
8.3.2 Record Functional Behavior (Binary Format) 145
8.3.3 Restore Functional Behavior (Deserialization) 147
8.4 Case Study 148
8.4.1 Evaluation Environment 149
8.4.2 Application of the Methodology 149
8.4.3 Performance Measurements 152
8.5 Discussion 153

9 Validation of the Performance Simulation Functions 157
9.1 Test Environment 158
9.2 CPU Performance Simulation Functions 158
9.2.1 Simulation of the Time 158
9.2.2 Open Loop 161
9.2.3 Closed Loop 163
9.3 Main Memory Performance Simulation Functions 165
9.4 Data Cache Memory Performance Simulation Functions 171
9.4.1 Validate Access Behavior 171
9.4.2 Validate Cache Set References 173
9.5 Simulated Software Functionality 186
9.5.1 Structure of the Test Application 186
9.5.2 Test Cases and Test Case Evaluation 187

10 Conclusion and Future Work **188**
10.1 Conclusion 189
10.2 Future Work 194

Tools Overview **198**

Abbreviations **200**

Bibliography **201**

List of Figures

1.1 Normal and Stubbed Execution with *Dynamic Performance Stubs* . . 4

2.1 Topics of Performance Engineering based on a Literature Review . . . 8

3.1 Memory Access Times in Cycles 26
3.2 Literature Review on CPU Caches 31
3.3 Example: 4-Way Set Associative Sector Cache 34

4.1 Interactions of *Dynamic Performance Stubs* 48
4.2 Framework of the *Dynamic Performance Stubs* 49
4.3 Classification of *Performance Simulation Functions* 50
4.4 Methodology for Using *Dynamic Performance Stubs* 54

5.1 Validate Component Under Study as a Bottleneck 68
5.2 Limited Bottleneck 73
5.3 Optimization Effort Estimation 76

6.1 Memory Simulation Algorithm 84
6.2 Calculating Appropriate Trace Points 92
6.3 Error in Simulation 96
6.4 Calculation of the *Intermediate Simulation Data* 106
6.5 Comparison of the *Intermediate Simulation Data* and *Simulation Data* 107
6.6 Measured Main Memory Behavior 108
6.7 Comparison of Original and Simulated Memory Behavior 109

7.1 Example Access Behavior of the *Data Cache Memory PSF* 120

8.1 Compare the Times Between Original and Stubbed Software Functionality 153

9.1 Simulation of Time Ranges 159

9.2 Global Open Loop CPU Stub Using a Dedicated Amount of Time . . 161
9.3 System Signal and Control Signal for the Closed Loop Algorithm . . 164
9.4 Total System Load Measured in the System while Closed Loop Algorithm was running 164
9.5 Time Spent in an Allocation Function Call 167
9.6 Different Memory Set Functions 168
9.7 Supposed and Measured Page Faults 170
9.8 Time Influence of the Malloc Pool 170
9.9 Validate Access Behavior of the *Data Cache Memory PSF* 172
9.10 Time Behavior of Register and L1 Cache Hits 175
9.11 Samples of L1 Read Misses / L2 Read Hits 177
9.12 Time Behavior of L1 Read Misses / L2 Read Hits 179
9.13 Time Behavior of L1 Write Misses 179
9.14 Influence of the Sets Parameter (excerpt) 182
9.15 Influence of the Arraysize Parameter (excerpt) 182
9.16 Samples of the L2 Read Misses for different Set Values 183
9.17 Generation of L1 Read Miss Events for Creating L2 Read Miss Events 184
9.18 Time Behavior of L2 Read Cache Miss Events 184
9.19 Structure of the Simulated Object 186

List of Tables

5.1 Determination of the Performance Behavior 74
5.2 *Flat CPU Stubs* Timing Measurements 75

7.1 Measured Memory Performance Behavior of the Original Function . . 135
7.2 Measured Memory Performance Behavior of the *Data Cache Memory Stub* 137

9.1 Trustworthiness of the Time Simulation 160
9.2 Portability of the Simulation of Time 160
9.3 Time Spent in an Allocation Function Call 166
9.4 Evaluates Different Memory Set Functions 168
9.5 Time Needed to Allocate and Load a Page 169
9.6 Cache Configuration for Validating the Access Behavior 171
9.7 Basic Configuration for the Validation of the Access Behavior 172
9.8 Configuration for Registers and L1 Cache Hits 175
9.9 Time Evaluation of the Registers and L1 Cache Hits 175
9.10 Configuration for L1 Misses / L2 Hits for Read and Write Access . . 176
9.11 Evaluation of Accesses and Samples 178
9.12 Time Comparision for Different Set Values 180
9.13 Time Evaluation for Read and Write L1 Data Cache Misses 180
9.14 Configuration for L2 Read Misses 181
9.15 Time Evaluation for L2 Cache Read Miss Events 185

Listings

5.1 Example Implementation of a *System Influencing CPU PSF* 65
5.2 Example Implementation of a *System Influencing CPU CF* 66
6.1 *Heap Main Memory PSF* 80
6.2 *Stack Main Memory PSF* 81
6.3 System Influencing Allocation of *Main Memory PSF* 82
6.4 Design of the Measuring Points' Data Structure 83
7.1 Example Architecture Description File 115
7.2 Example *Simulation Data* File 116
7.3 Algorithm to Specifically Access Different Cache Sets (excerpt) 117
7.4 Function to Specifically Access Different Cache Sets (excerpt) 118
7.5 Example Architecture Description File to Create Level One Hits . . . 121
7.6 Caching Architecture 133
7.7 *Simulation Data* File (Case Study) 136
8.1 Example: Serialize a Fixed Sized Integer Array, which is inside of a C++ Class 144
8.2 Stores a Data Structure (Class) 145
8.3 Example: Decoded and Semicolon Separated Trace File Entry 147
8.4 Restore Functionality of the *libSSF* 148
8.5 Excerpt of the Class "Connection" 150
8.6 Example of Serialized Class Member 150
8.7 Serialized "Connection" Object 150
8.8 Serialization Specification of the "Connection" Object 151
8.9 Excerpt of a Decoded Trace File 151
9.1 Simulation of a Duration 159

Chapter 1

Introduction

This chapter presents a novel approach to support the methodologies of software performance engineering with the possibility to realize a cost-benefit analyses of a proposed software performance improvement. Following the motivation, the concept of dynamic performance stubs is introduced. A general overview of the framework is provided and the application of the stubs to carry out a cost-benefit analysis is presented. Furthermore, an outline of this thesis is given.

Motivation

> *"Optimization matters only when it matters. When it matters, it matters a lot, but until you know that it matters, don't waste a lot of time doing it. Even if you know it matters, you need to know where it matters."* [87]

The methods of Software Performance Engineering (SPE) [97] are used to determine the software functions, which have to be optimized. The basic concept of the methods used for the performance measurement and performance tuning process, which are a subset of SPE, follows a strict line: analysis - test - improvement - verification (summarized from [81]). This procedure strongly depends on the experience of the person realizing the performance optimization study and can roughly be used for estimations of the achievable gain [81].

This often leads to an over- or under-optimization of the software module as the maximum gain cannot be determined in advance.

In the first case, the software module is highly optimized without getting the expected improvement as another bottleneck appears in the system [55, 84]. Here, usually too much effort is spent to optimize the module.

In the other case, the realized optimization does not fully utilize the systems' resources, and thus, performance improvement capabilities are wasted. Hence, another function, which can be improved has to be determined and optimized. This needs additional effort, which has to be spent.

Problem Statement

One problem by using the methodologies of SPE is that a cost-benefit analysis of improvement possibilities can rarely be done. The expert has to know the complete system at a very detailed level, which is hardly possible within large software projects.

Another problem is that the optimization has to be done without really knowing how much of the module has to be optimized [84].

Without knowing all bottlenecks in advance, the improvement effort may lead to an unexpectedly small gain. So, the following questions arise: "How much performance improvement gain can be expected by optimizing this software function?" and "How much effort has to be spent for reaching this improvement gain?"

These can be answered by a cost-benefit analyses, which is not available within the common methods of performance measurement and performance tuning process. To bypass these problems, a new SPE methodology has to be developed to realize a cost-benefit analysis in advance of the realization of the optimization. This can be achieved by simulating different optimization levels of the software function and by evaluating the performance improvement achievable in the system.

Solution Approach

This section describes a novel approach to solution, called *dynamic performance stubs* (DPS). They provide a framework and methodology to carry out a cost-benefit analysis for a software optimization studies.

The objectives are to develop a well-found methodology to simulate different performance optimization levels of software modules or functions, and, to evaluate the outcome of a possible optimization in advance of a costly implementation of the optimization.

Dynamic Performance Stubs

DPS combine the techniques of stubbing, used in software testing [6, 109], and the concepts of performance measurements and tuning [35, 42, 55, 74] from SPE. This combination is reflected in the *dynamic performance stubs* framework. Here, its two major elements are the *simulated software functionality* (SSF) and the *performance simulation functions* (PSF):

SSF As the *DPS* are used to simulate the performance behavior of a software bottleneck within a real application, the functionality of the software module has to be rebuilt with only a small overhead of time. This is achieved by applying the *SSF*.

PSF The *PSF* are used to adjust the performance behavior of the *DPS* in order to be able to simulate different optimization levels of a software module or function.

By using those two elements, a *DPS* can be built, which replaces the software functionality of the supposed bottleneck. This can be achieved by using a predefined set of test cases as the functional behavior of the bottleneck has to be deterministic.

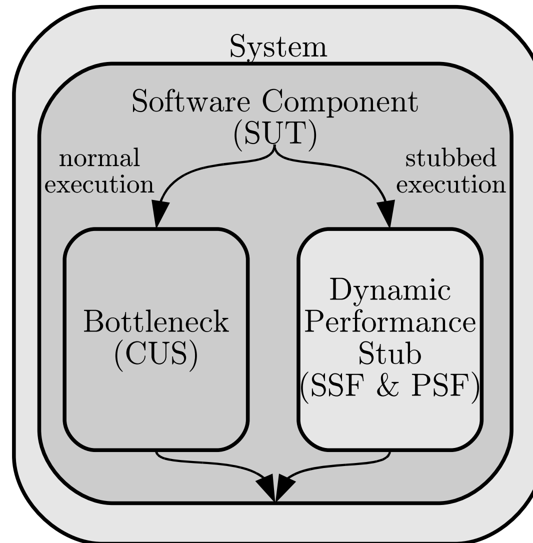


Figure 1.1: Normal and Stubbed Execution with *Dynamic Performance Stubs*

Figure 1.1 depicts both, the original execution of the system as well as the execution where the bottleneck (component under study, CUS) has been replaced by a *dynamic performance stub*.

The software performance test can be executed several times by changing the performance behavior of the *dynamic performance stub*. The results of the tests present the overall performance improvement gain depending on the adjusted performance parameters of the *dynamic performance stub*. Moreover, they can be used to do a cost-benefit analysis of different performance optimization strategies.

Additionally to the benefits of the *dynamic performance stubs* many drawbacks, which a software performance improvement usually have, can be reduced, e.g., an over- or under-optimization can be avoided if possible. Moreover, the following non-exhaustive list provides vantages, which can be furthermore achieved:

- Identify hidden bottlenecks
- Evaluate the work needed for optimizing the CUS
- Ability to identify the performance bound
- Evaluate changeovers of performance bounds
- Evaluate the amount of work needed to reach a changeover of the performance bounds

This is only a short overview of several advantages, which can be gained by using *dynamic performance stubs*. The elements of this list as well as a description to achieve these benefits are evaluated more in detail in Section 4.

Thesis Outline

This thesis is structured as follows. The next two chapters (Chapter 2 & 3) present a critical review of literature in the areas of performance engineering and, especially, *dynamic performance stubs*.

In Chapter 4, we present the framework of the *dynamic performance stubs*. This chapter serves as an introduction to the new developed methodology for software performance engineering. Hence, an overview of the *dynamic performance stubs* is presented and subsets are specified. General steps to use the *dynamic performance stubs* framework in order to optimize software performance bottlenecks are also presented. An extension of these steps to evaluate further use cases and to increase the usability of the *dynamic performance stubs* are given. Finally, the section concludes with a summary.

The Chapters 5, 6 and 7 describe three different subsets, i.e., *CPU stubs*, *main memory stubs* and *data cache memory stubs*, of the *dynamic performance stubs* in detail. Each of these chapters lists the requirements of the *performance simulation functions* and defines a methodology to apply these. Moreover, we provide an implementation of the *performance simulation functions* as well as a possibility to calibrate these functions to the system. To conclude each chapter, a case study is presented and followed by a summary.

Chapter 8 discusses a framework to simulate the functional behavior of a software algorithm. Thus, the requirements on the *simulated software functionality* are listed. A methodology that describes the application of the *simulated software functionality* is presented. Furthermore, a realization is depicted and followed by a case study as well as a summary.

To support the approaches of the Chapters 5 - 8, Chapter 9 provides an evaluation of the usability of the *performance simulations functions*. Hence, we shortly introduce the test environment, which is used for validation. Moreover, this chapter is split into the different subsets to evaluate each *performance simulation function* separately.

In Chapter 10, the scientific results and possibilities to extend the framework of *dynamic performance stubs* are presented.

Chapter 2

Literature Review on Performance Engineering

This chapter provides an overview of the different performance engineering aspects. Especially, an overview of the history, the different areas of performance engineering and performance engineering studies are presented.

2.1 Introduction to Performance Engineering

Performance Engineering (PE) includes different aspects, e.g., performance requirement analysis, performance specification, capacity planning and -management, performance modeling and performance evaluation[1]. A sustainable work of PE has been done in software performance engineering (SPE).[2] SPE is a systematic approach to evaluate and validate the performance behavior of software in order to meet the performance requirements. Therefore, the methodologies of SPE are targeting many areas, such as performance predictions of upcoming software, performance measurement and improvement studies. Another field of SPE is the performance requirements analysis and specification. While a software system has many different performance indicators, software performance is often referred to throughput or latency. Performance regarding software engineering has been defined by IEEE[49]:

> *"**performance.** The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."*

The work of a good performance engineer is often not recognized because if they are doing their job well, there are no performance problems. So someone might think: "Why do we need a performance engineer if we do not have performance problems?"[106]

This section gives an overview of the history and the different aspects of performance engineering as depicted in Figure 2.1. Additionally, it introduces performance measurements studies and validation methodologies. It is concluded by describing performance measurement tools.

2.2 History of Performance Engineering

Software performance is considered since the beginning of computing [104]. Around 1968 Donald E. Knuth worked in the area of efficient sorting and searching algorithms as well as data structures [66, 67]. Whereas, many authors worked on performance-oriented development approaches, references can be found in [103, 104],

[1] This list only provides some aspects of performance engineering and is far from being exhaustive.

[2] Since SPE now covers almost every aspect of PE it is often set to be equal.

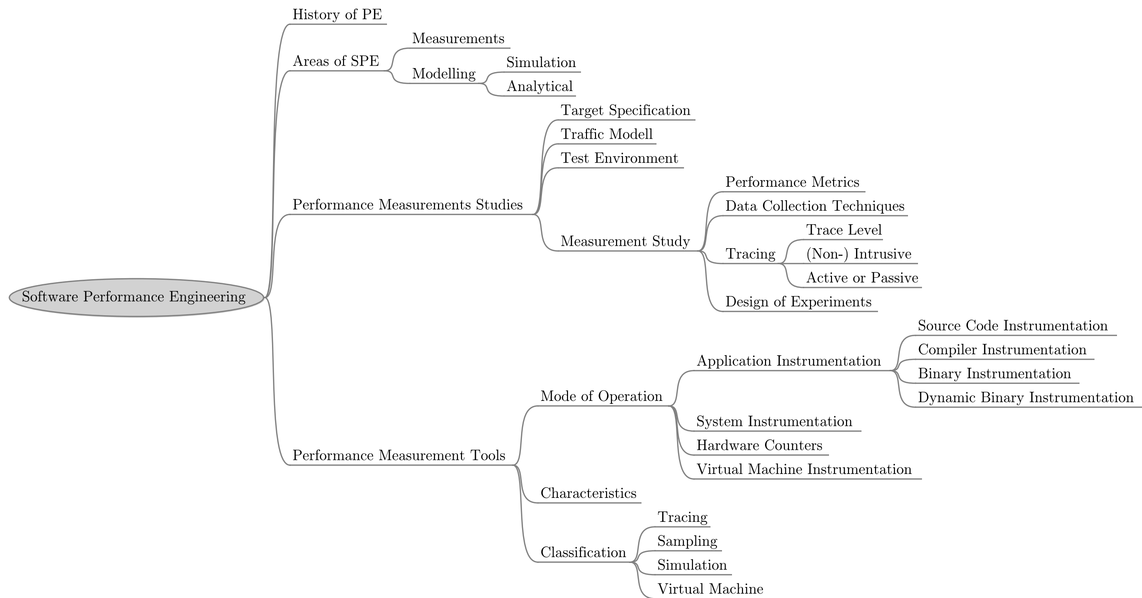


Figure 2.1: Topics of Performance Engineering based on a Literature Review

around in the 1970s most software developers of non-reactive systems used the “fix-it-later” approach for tuning their software. The hardware has developed rapidly so that the “fix-it-later” approach was sufficient then. In the 1980s, the complexity of software systems increased dramatically. Therefore, the demand for “fast” software also increased drastically. [45], published in 1984, lists more than 200 references for performance evaluation methodologies focusing on post-1970 developments and trends. In 1981 the methodology of SPE has been introduced [101, 104] and updated in [102, 105]. More recent articles [9, 103, 104] and books [35, 55, 74] about software performance methodologies provide sustainable work in that area.

2.3 Areas of Software Performance Engineering

SPE includes all areas of software performance predictions, such as predictions about the performance behavior of non-existing soft- and hardware as well as predictions of upcoming possible workload scenarios. Another part of SPE is the performance measurement and improvement studies. A more comprehensive list of SPE techniques can be found in [104]. According to the following articles [35, 45, 58, 103] SPE can be split into the different parts:

- Performance Measurements
- Performance Modeling

  - Simulation Performance Modeling
  - Analytical Performance Modeling

A fourth area of SPE ("Petri Net Modeling") is mentioned in [35]. Because, it is more a technical realization to support a mixture of "simulation-" and "analytical performance modeling" it is not included as an independent category in this thesis.

2.3.1 Performance Measurements

Performance measurements are used to evaluate and validate the performance behavior of existing systems or applications. This is a crucial task as performance predictions of software often do not meet the performance specifications. Performance measurements of existing systems or prototypes are often used to provide a more detail view to support model specifications, to validate performance objectives and to identify performance problems that require improvements. The task of a performance engineer and several different evaluation methods and techniques are described more in detail throughout the following subsections.

2.3.2 Performance Modeling

Whereas, performance measurements are done to validate the behavior of applications performance modeling will be used for non-existing systems, to support performance measurements or when performance measurements would be too expensive. According to the literature, e.g., [58] performance modeling can be split into simulation performance modeling and analytical performance modeling as described above.

Simulation Performance Modeling

Simulators model existing or future hardware to evaluate different realization possibilities. The simulators are often applications running on existing hardware and model the proposed behavior of this or another system. They are working instruction accurate and can provide deep insights into the functionality of the application or hardware. In this case, the execution of the SUT takes normally much more time as on a real system. Simulators are also often used to evaluate timing behaviors, too. As many different models can be realized, these tools can simulate complete systems or only one or more components. A drawback is that simulators often provide many

details of the system and, therefore, imply too much accuracy. Insufficiencies occur if the component under test is not accurately simulated, e.g., a least recently used (LRU) replacement strategy is used instead of a pseudo least recently used (PLRU) is used. Hence, having detailed results do not necessarily mean that the results are appropriate.

Analytical Performance Modeling

This modeling technique is mainly used to evaluate the performance behavior of large systems. These systems can only partly be analyzed using cycle accurate simulation performance modeling tools. The analytical modeling is based on mathematical descriptions and, therefore, these tools are often cost and time effective. As the "real world" is too complex to fit into models, the tools are based on simplifying assumptions. Hence, all results are only approximations to the "real world" behavior. Nevertheless, carefully constructed analytical models can be used to more or less accurately evaluate "average job throughput", device utilization and response times. Many valuable references to analytical performance modeling can be found in [45].

2.4 Performance Measurements Studies

The design of the performance measurement study is one of the key aspects for any software optimization process. These studies are normally done in a very late stage of the performance engineering process [58, 71, 87]. Lists of software improvement possibilities can be found in [1, 104].

The performance optimization process should be integrated into the software development life cycle [58, 71, 106]. In many software projects, e.g., in embedded systems or telecommunication software, the development platform is often not the target platform.

For the ease of use, the optimization team should provide performance measurement environments to the development team. Therefore, they have to identify suitable performance measurement tools and have to validate the usability of these tools on the development platform. Having this, the development team can optimize identified bottlenecks by themselves and measure the estimated improvement gain. The final results have to be evaluated on the target platform, of course. Additionally, performance tests on the final target are more expensive than evaluations on the development platform, since the target hardware is often specialized. Therefore,

providing a performance measurement environment to the development team can reduce costs and speed up the whole optimization process.

2.4.1 Performance Target Specification

A performance target specification (PTS) will normally be written in the system design phase and includes several different hard- and software configurations of the system. It specifies performance indicators (performance metrics) as well as the according workload scenarios and test conditions. Therefore, it includes the performance targets of the system and can be used as reference for performance improvement studies. Despite of the functional specification that describes several use cases of the software, the performance target specification lists non-functional requirements of the system.

The following paragraph from the introduction of the performance target specification of our industrial partner for their telecommunications system is a good example of the content of a performance target specification:

> *"If not specified otherwise, end-to-end performance objectives are defined in terms of Performance Indicators (PIs), e.g. latency and throughput pertaining to a single user in an empty (unloaded) system under the most favorable conditions, e.g. ideal radio conditions, so that they can be tested in a System Verification lab or in a customer lab with commercially available test equipment."* [76]

The information provided in a performance target specification document are often on a detailed level, e.g., the time needed from message request until response. Nevertheless, the targets have to be broken down further by the performance group for identifying performance bottlenecks.

2.4.2 Traffic Model

Together with the performance target specification a traffic model is often evaluated by the system design performance group. This model describes workload predictions for the new system and is normally based on measurement of similar former systems and extended with different workload predictions and possible use cases. This document is often split into different traffic type perspectives describing particular use cases, e.g., the traffic model of our industrial partner [77]:

Traffic Needs The "traffic needs" in general divides the needed traffic appearances according to their specific use cases, e.g., user initiated traffic or configuration data.

Traffic Types The "traffic types" describes the aspect of network connections, e.g., multicast or point-to-point data flows.

Traffic Characteristics The "traffic characteristics" describes the amount of different "traffic types" as well as their appearances and use cases.

A summary for a chapter from the introduction of the traffic model can be used as an example for the content of a traffic model in general:

> *"The possible "mixes" i.e. how to utilize the created Traffic Model parameters starting from basic user plane traffic load issues by choosing a reference "call" as a simple base (reference call mix). The reference call mix assumes static user plane load to the system based on one single subscriber "type" using the most common service in the system. The related Control Plane load issues for the chosen reference call can be gathered via the related service mix parameters. When using different subscriber types (utilizing subscriber mix), different types of services (via different service mix models) and taking into account the impacts from the network structures (connectivity mix) the traffic model parameter utilization gets more sophisticated but gets also more complex."* [77]

This example is derived from the evolved NodeB component of the telecommunication system LTE (see [65]). As it is a telecommunication system, many different use cases regarding a "call" exist, e.g., voice call or data call. Therefore, the traffic model describes the different call scenarios, e.g., reference call mix. These scenarios are further classified and described.

As this example shows, the specification of a traffic model is a highly complex task with many uncertainties, e.g., predictions for the upcoming traffic and use case scenarios. Hence, the traffic model specifies different workload scenarios. The most called procedures can be extracted and used for performance optimization studies.

2.4.3 Performance Test Environment

Most requirements for functional software test environments are also applicable to performance test environments, e.g., being able to create deterministically and re-

producible test results. In addition, to these requirements, performance test environments should be able to almost fully utilize the system under test.

Another key requirement for performance tests is the possibility to simulate use cases that are specified in the performance target specification as well as in the traffic model. Performance test cases often differ from the functional test cases [106]. Sometimes test cases from the functional tests can be used to measure performance targets specified in the performance documents but often the test cases have to be adapted to fit the needs for a performance evaluation study.

2.4.4 Measurement Study

There are several types of performance testing, e.g., capacity-, stress- or long time tests. Whereas, each of these tests has it is own specialization this section is more generalized and provides an overview of different decisions, which has to be targeted in a performance measurement study. This section discusses the following aspects:

- Performance Metrics
- Data Collection Techniques
- Tracing
- Performance Measurement Tools
- Design of Experiments (DOE)

Performance Metrics

Based on the performance target specification (see Section 2.4.1) and traffic model (see Section 2.4.2) the performance metric has to be chosen. Typical performance metrics are and defined according to IEEE [49]:

> *"**response time.** The elapsed time between the end of an inquiry or command to an interactive computer system and the beginning of the system's response."*
>
> *"**throughput.** The amount of work that can be performed by a computer system or component in a given period of time; for example, number of jobs per day."*

> *"**utilization.** In computer performance evaluation, a ratio representing the amount of time a system or component is busy divided by the time it is available."*

A more comprehensive list can be found in [48].

Data Collection Techniques

In order to verify the performance behavior of the system, the performance metrics have to be evaluated. This can be done by either monitoring or by recording the events.

Monitors are normally used to evaluate the system behavior during the runtime (online). To verify the application behavior a recorder can be used to store the performance metrics. These traces are often evaluated afterwards, which is also referred to offline evaluation.

Depending on the performance metric, a system wide or a program monitor can be used. Different events of the application can be recorded with either a system event recorder or using an external program event recorder. As both mentioned types of recorders only provide an external view another type of recorder, called internal event recorder, is available to collect different performance data or metrics. [103] lists several different performance metrics as well as the according monitors and recorders.

Tracing

Tracing is a type of logging and can be used to evaluate the application behavior. Usually, debugging information will be recorded by traces to get insights of the program execution. Additionally, this method is often used to record information about the performance behavior. The following subsection discusses different aspects of tracing more in detail.

Trace Levels It is often necessary to add additional performance trace information to the application as well as switching off unneeded debug trace information. This can be realized by different trace levels. This is needed to improve the runtime behavior of the application as every trace information contributes to the overall execution time. Therefore, unneeded trace information are often removed, e.g., using flags, during performance test.

(Non-) Intrusive The measurements can be either intrusive or non-intrusive [9]. Intrusive here means that additional trace probes are inserted in the workload and will explicitly be measured, e.g., sending messages with non-existing data content to be able to uniquely identify the received message. This can easily be used for measuring, e.g., round-trip times. Non-intrusive measurements do not add messages to the workload. All information needed must be extracted from the workload. As they are non-intrusive, they will not influence the overall workload of the system.

Active or Passive Sometimes traces cannot be received by the application itself. This is often the case if the system has to be tested under high workloads, e.g., for measuring overload routines. Therefore, it is necessary to passively measure the SUT. Passive measurements, despite of active, will be taken from outside of the software or system, e.g., using a mirror port from network switches to gather information about the network traffic. These measurements do not influence the system or software and, therefore, often lead to results that are more accurate. Nevertheless, passive measurements cannot provide deep insights of the system or software under test because this type of information can only be recorded inside of the application.

Performance Measurement Tools

Depending on the performance metrics and the test environment a suitable performance measurement tool has to be chosen. More information about different tools and recording techniques are discussed in Section 2.5.

Design of Experiments

There are normally different optimization possibilities that can be used to improve the performance of a software and system. These optimizations often depend on each other. As every improvement has some drawbacks [55, 84, 87, 97], it is useful to evaluate the impact of each optimization. This can be done by defining a test matrix, e.g., the different optimization possibilities and the according experiments. Having this matrix the performance engineer has to decide, which of these possible combinations shall be evaluated. This is called “design of experiments” (DOE) and is described in [45, 134].

An overview classification of the DOE as discussed in [55] lists the following main categories:

Simple Designs In simple design experiments common configurations will be used and only a single parameter at a time will be changed (one-factor-at-a-time).

Full Factorial Designs Fully factorial designs is the opposite of simple designs. In this case, every possible combination of varying the parameters will be evaluated. The best combination of parameters can be evaluated. Of course, in this validation scheme the most effort has to be spent.

Fractional Factorial Designs Fractional factorial design is a combination of the above described designs to reduce their drawbacks. Here, different factors will be changed for evaluating the behavior. The choice of the factors, which will be evaluated, is a difficult task. Choosing the wrong parameters can lead to useless results.

In [55] several chapters are discussing the design of experiments.

2.5 Performance Measurement Tools

In this section, performance measurement tools are described more in detail. There are many different available tools for many different performance analyzing possibilities. They are described in software performance books, such as [35, 55, 74]. This section concentrates mainly about to measuring executable applications, which is also often referred to "dynamic analysis" (see [86]).

2.5.1 Mode of Operation

Performance measurement tools as described in Section 2.5.3 are distinguished on the mode of operation: "tracing tools" and "sampling tools". In order to collect the different performance metrics they have to utilize the system. This can be done by instrumenting (see [86]) either the source code, the binary or the system by inserting probes.

A metric is called "directly" if it itself can be measured [74]. Here the instrumentation is often done in the application. This technique can be used to gather detailed internal application information. Another possibility to gather performance metrics is to measure "indirectly". In this case, another performance metric is measured and conclusions to the performance metric under study will be drawn. This delivers an external overview of the metric. Introducing probes in the application always

affects the behavior of the execution. The measurement error depends strongly on the probes itself. Probes can be inserted in the following parts of a system:

- Application Instrumentation
- System Instrumentation
- Hardware Counters

These items will be explained in more detail in the remaining section.

Application Instrumentation

The following list presents several different possibilities to instrument an application:

- Source Code Instrumentation
- Compiler Instrumentation
- Binary Instrumentation
- Dynamic Binary Instrumentation

Source Code Instrumentation This is a main technique for gathering performance data and is normally done by the developers. The advantage of this method: the needed information can be measured from inside of the application with normally small overhead. The main drawback of this method is that someone will not be able to find the real bottleneck if the application is not correctly instrumented. Additionally, deep insights of the application are essential to get the needed information.

Compiler Instrumentation The instrumentation will be done while the source code of the application is compiling. Most modern compilers provide a possibility to interact with profiling tools, such as GPROF or Rational Quantify. These tools are providing options for automatically inserting probes. Often the collected traces after a performance measurement run can be fed in the compiler for a better optimization, e.g., rearranging the instructions for fewer branch miss predictions in this special case.

Binary Instrumentation The source code is needed for applying source code or compiler instrumentation methods. A binary instrumentation can also be done without available source code. It analyzes the binary code and introduces the necessary probes. Examples for such tools are TAU or EEL.

Dynamic Binary Instrumentation The main difference between "dynamic binary instrumentation" and "binary instrumentation" is the point in time when the binary gets instrumented [86]. In "binary instrumentation" the binary is first instrumented and afterwards executed. In dynamic binary instrumentation the binary is instrumented at run-time. This can be done with an external process or with a process hooked into the binary. Examples for such tools are TAU, PIN or paradyn.

System Instrumentation

The probes for the performance measurements will be inserted in the operating system or kernel. This can be achieved by extending the system by another measurement application that additionally often instruments the system itself. The kernel has often to be patched and rebuild for including the probes. This technique is able to indirectly trace the application. Examples for such tools are: LTT, LTTng, LKST or Systemtap.

Hardware Counters

The processor architecture often provides many hardware counters [50, 79], e.g., "instructions retired" or "L2 data cache miss". They can be used to analyze different performance behaviors. To gather the information additional applications, such as OProfile or Perfctr, are running in the kernel. The counters only report an overview of the complete system. In order to get reliable results many samples have to be captured. Collecting and gathering the values from the counters does not cause much overhead. Therefore, long-term application runs are possible and often necessary due to the measurement error introduced by the sampling technique, which is also described in Section 2.5.3.

Another possibility is to insert probes into a "virtual environment" or an emulator. In this case, the application under study has to be executed within this environment. This is similar to using a simulation tool to measure the application under study.

2.5.2 Characteristics

As every software application, performance measurement tools should comply with many different characteristics. This section will shortly outline the main requirements dedicated to measurement tools. A more comprehensive list of the requirements of performance measurement tools as well as performance evaluation techniques are described in [58]. Performance measurement tools should introduce as little overhead as possible. They should try to minimize the measurement error and should provide deep insights of the system or application under test. Additionally, they should be highly configurable and easy to use. As there are many competing requirements for performance measurement tools, a diversity of different types of tools have emerged.

2.5.3 Classification

Whereas, most performance evaluation tools cannot be strictly assigned to a dedicated category, the different ideas and methodologies of the measuring possibilities have been classified in the literature [45, 58, 74, 82]. The commonly used breakdown is as follows:

- Tracing Tools
- Sampling Tools
- Simulation Tools

There is another category of performance measurement tools called "benchmarks tools", e.g., SPEC CPU2006. They are exhaustively described in the literature [22, 58]. Benchmarks are used to create a defined workload and to store the according performance results. As the workload is defined the results of the benchmarks for different software or systems can be compared. Benchmark tools can be used to identify software bottlenecks. As these tools do not provide deeper information about the software under test, they will not be described furthermore in this section.

Tracing Tools

Tracing means measuring the execution path of a running application. Normally, the application is instrumented to get the necessary performance and runtime information. This is done by the software developers but can also be done by performance

applications (see Section 2.5.1) or by compilers. Additionally, the system can be instrumented to get an overview of the running threads and application in a multi-threaded environment, e.g., using LTT or LTTng.

This measurement method can be used to get deep information about the process but the probes should be chosen carefully. As every trace information contributes to the total run time of the application. Therefore, tracing can be very expensive for long time runs with many different probes.

Sampling Tools

Sampling or profiling is the periodically capturing of the state of running applications [45]. In contrast to the tracing tools, where the flow of an application can be recorded, sampling tools normally only provide statistical behavior of the process or application, e.g., determining the number of function calls. This performance information is often provided by hardware performance counters [16, 140]. Tools, such as Systemtap, OProfile, Perfctr or Dtrace can be used to gather these data. Another possibility for sampling-based measurement is to automatically instrument the application or to manually instrument the application, which can be done by the developers. For more information see Section 2.5.1.

Sampling normally means that the system counts an amount of events and records the values at fixed time intervals [74] (“time based”). Another possibility is to capture an amount of events and as soon as the value reaches a threshold the value will be recorded (“event based”). Both sampling modes, “time-” and “event-based”, are supported by OProfile (see Section 4 of the OProfile manual). Additionally, sampling techniques can use further triggers, e.g., precise event-based sampling (PEBS) [16].

Simulation Tools

Simulation tools are simulating software or hardware where the application under test can be executed. They provide often deep and accurate information in their dedicated areas. Unfortunately, they are slow compared to the real execution of the application on the target.

Callgrind[3] is a good example for this category. It is able to simulate a complete system, including different memory architectures and is able to trace and profile the whole application. It provides an execution path as well as profiling data, e.g., L1

[3]Callgrind is an extension to valgrind. See `http://valgrind.org`

cache instruction read misses and hits. If the source code is provided and debug symbols are available in the application than kcachegrind[4] is able to provide information down to the assembler instructions. Due to the fact that callgrind simulates the whole architecture it is not possible to get information of the amount of cycles spent in the application. Since it only models the architecture most of the values are only based on approximations or assumptions, e.g., the L2 cache hardware prefetcher is able to load all data. Another drawback is the time needed to measure the application: callgrind needs 20 - 100 times longer then a normal execution[5]. For more information see [53, 86] as well as their research paper section[6].

Since use of virtualization of computer hard- and software is increasing in the last years, many performance tools have been built on top of these techniques. Therefore, we would like to add another category "virtual machine tools" to the list above. These tools are similar to the ones described as simulation tools. However, virtual machine tools do not really fit into this category since simulation tools are often simulating a whole system in total (including the CPU), which is a very time consuming task.

Summary

In general, different measurement tools provide more or less accurate and deep insights of the software or system. The more information are recorded, the more time is needed for recording as well as the more accurate the information are the more time is needed. The time spent in a measurement tool changes the software behavior. Therefore, the performance measurement tools and metrics have to be chosen carefully.

[4] A visualizer for callgrind traces. `http://kcachegrind.sourceforge.net/html/Home.html`
[5] `http://valgrind.org/info/tools.html`
[6] `http://valgrind.org/docs/pubs.html`

Chapter 3

Literature Review in the Area of Dynamic Performance Stubs

This chapter evaluates current literature regarding dynamic performance stubs as well as the subsets CPU- and memory stubs. Moreover, literature in the area of the simulated software functionality is presented.

3.1 Dynamic Performance Stubs

In [82, 83] the performance is modeled at instruction level as well as the influence on memory and caching performance. The granularity makes the approach hardly usable for *dynamic performance stubs*, which will be used to stub modules or functions.

In [92] *smart stubs* are introduced. These stubs simulate the budget regarding storage and timing behavior of systems. These systems do not exist while the measurements are done. Also mainly a management view has been taken. The *dynamic performance stubs* however aim to replace a known software bottleneck to evaluate different optimization possibilities.

However, the *dynamic performance stubs* in our approach will be used for stubbing already implemented and measured software parts in order to find the bounds of the performance improvement within that part. This procedure helps to identify the real gain of the performance improvement without really improving it and additionally shows the next bottleneck. So the cost-benefit analysis for improvement activities can be achieved in a more realistic way, because a proper simulated result is better than a simple estimation.

To the best of our knowledge, no research in the area of replacing software bottlenecks by stubs that can simulate different software performance behaviors has been done. Many performance engineers, whom we have met, have confirmed that no such methodology or tools are available.

3.2 CPU Stubs

CPU stubs refer to simulating the performance behavior of a software module or function regarding the CPU. Hence, the time behavior of a process has to be studied and simulated. This can be done by simulating the dedicated amount of time while the process is scheduled.

3.2.1 Related Work

In [130, 131] a problem with simulating a dedicated amount of time with do-nothing loops is described. Despite the problems seen, there are big differences in the approaches. The procedure is targeting the area of bulk-synchronous parallel jobs, which are realized as do-nothing loops. The focus is to optimally utilize each of

the included processors. So the processes always try to run, ignoring the amount of time needed for the operating system per processor. As soon as the operating system has something to do, the userspace application will be scheduled out and the total execution time will be delayed.

Our *CPU performance simulation function*, however, will be calibrated in an otherwise idle system with enough time for the operating system. As experimentally proved in [122], in our environment the execution time of a process can be simulated with a do-nothing loop, predictably in contrast to [130, 131]. Additionally, because of the fact that such a loop has a defined number of instructions these loops can be used to simulate the time behavior of processes.

3.2.2 Basic Literature

Process States The CPU executes the instructions of the applications. The scheduler exists to decide which process the CPU has to execute next (see [8, 36, 116]). It maintains several queues about the states of all processes: running, ready or blocked. If no process is either in the running or ready state the idle process is executed by the CPU.

From a process perspective, the process can be either executed by the CPU or is suspended. *CPU stubs* are targeting to simulate the timing (CPU) behavior of a bottleneck. Hence, *CPU stubs* have to be able to switch the state. As only the scheduler decides whether a process is in the running or ready state. Thus, it is not possible to enforce a process to be in the running state in non-real time systems acting from user space side. Only the possibility to increase the chance to be scheduled soon into the running state exists, e.g., using priorities.

Therefore, the *CPU stubs* have to simulate the remaining states “running” and “ready/blocked”.

Types of Time Several different types of time have to be distinguished to describe CPU behavior of a process. According to [64, 116], those are:

User Time This is the time spent in the process while it is executed in the userspace.

System Time This is the time spent in the process while it is executing systemspace routines, e.g., allocating memory.

Real Time This is the time, which the process was active, i.e., from starting the application until its termination.

As $usertime + systemtime \neq realtime$, there is another time period, which has to be considered [64]. This time is called “wait time” and describes the time while the process is blocked by any reason, e.g., waiting for I/O or because the CPU is blocked by another process.

The before mentioned times present the times, which a process can have. From a CPU utilizations perspective, two different time exist:

Idle Time This is the time while the “idle process” is scheduled.

Busy Time This is the time while a process is scheduled, which is not the “idle” process.

These two time can be used to calculate the CPU utilization.

Measuring Time As described above, different types of time can be measured. First of all, there is the total time spent in the system. Architectures normally provide two devices for time keeping [75].

The first is the real time clock (RTC), which is system independent and keeps track of the absolute time. It is normally used by the kernel to initialize the wall clock time but can also be read by other applications [8].

The second is the time stamp counter (TSC) [2, 31]. It is updated with each tick of the system and can be used to measure the total time passed in the system. These are possibilities measure to the total time accurately. For a more granular and portable wall clock time measurement the POSIX standard[1] specifies several functions such as the “gettimeofday()” (sys/time.h; time.h) or the “time()” (time.h).

Linux also provides the possibility to measure the time used by the dedicated process. For this purpose the “clock()” (time.h) or the “times()” (sys/times.h) function can be used. Also more detailed information can be gained by using the “getrusage()” (sys/time.h; sys/resource.h).

In some cases, it cannot be possible to measure the needed time period, e.g., because of the resolution of the timer [74]. Especially, if the measured events are smaller or at the same size as the resolution of the timer. According to [74], one possibility to measure these time periods is to make many successive measurements of the event within a single measurement. Here, a statistical estimation of the events duration can be done.

[1] http://www.pasc.org/plato/

According to the needs of the performance analysis one or more of the above mentioned measurement functions fits more or less well. Therefore, the most suitable function has to be chosen.

CPU Utilization Beside the time measurement, it is often useful to measure the utilization of the CPU usage. There are several dedicated applications for a system wide measurement, just to name two: "top" and "vmstat". To measure the CPU utilization from inside of a module or function the "proc"-filesystem can be used. Here, the "stat"-file provides a global view, e.g., of the CPU utilization. To measure only self-used information within the process, the "stat"-file which is located in /proc/⟨*pid*⟩/stat can be used.

3.3 Main Memory Stubs

Main memory stubs shall be able to simulate the main memory access behavior of software modules or functions. They can be used to replace a memory bottleneck in order to easily change the performance behavior to evaluate different performance optimization gains.

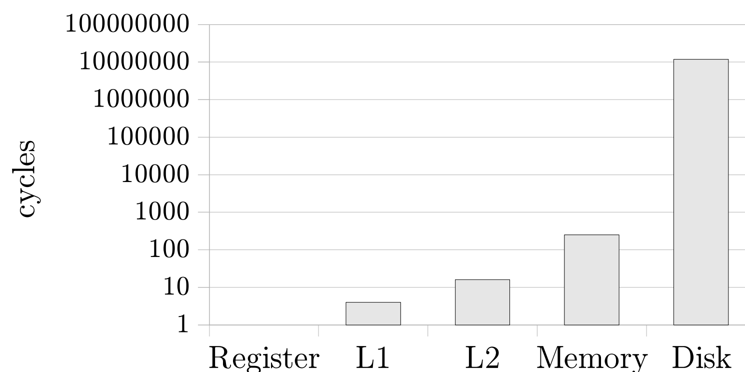


Figure 3.1: Memory Access Times in Cycles

Figure 3.1 depicts the access times to different memory levels in terms of cycles. The y-axis is scaled logarithmic. The values of an Intel Pentium 4 processor architecture have been used in this diagram.

3.3.1 Related Work

Performance skeletons, as described in [108], are used to simulate the performance behavior of applications. This has been achieved by capturing the execution be-

havior and creating synthetic program skeletons. Hence, the skeletons are using an instruction mix, which is similar to the instructions used by the application under study. The main target is to simulate the performance influence of the application in long running scenarios by short time execution of the skeletons. This can be used to accurately estimate the performance in heterogeneous and shared computational grids. In [115] the automatic constructions of these skeletons is shortly described. The possibility to replicate the memory performance behavior is studied in [119]. The reader is referred to [107, 133, 137] for more information about performance skeletons.

The approach of the *dynamic performance stubs* framework significantly differs from the performance skeleton approach. The performance skeletons are used to simulate the performance behavior of all performance bounds concurrently, e.g., cache and main memory, and only aim on the reduction of the running time of the application. In our approach, we simulate each performance bound independently, to be able to separately adjust each performance behavior to the needs of the performance improvement study. This leads to a gain-oriented optimization. Moreover, the *dynamic performance stubs* framework is able to simulate a function or module. Whereas, the performance skeletons are always simulating the behavior of the whole application. Therefore, the approach of the performance skeletons cannot be applied to the *dynamic performance stubs* framework to simulate the memory behavior.

3.3.2 Basic Literature

This section is based on a literature review for memory handling in the programming language C. Furthermore, it provides some basic understanding of the memory layout of applications.

Memory Usage in Computer Systems An application typically consists of five memory segments [99]: Code, Data, Block Started by Symbol (BSS), Heap and Stack.

The code segment, also known as text (segment), is a portion of memory, which stores the instructions executed by the application. The next two segments, i.e., data and bss, stores variables, which are allocated during the compile time. The heap segment stores variables, which are dynamically allocated during run-time of the process. The stack memory is used to store temporarily used variables, e.g., within function calls. Hence, the stack memory is more often allocated and freed than the

heap memory. Additionally, the stack usually allocates only few bytes, whereas, the amount of bytes allocated on the heap is higher. Moreover, the stack only de- and allocates the data on top of the memory, whereas, the heap memory always tries to allocate the data in an appropriate memory region. Hence, fragmentation of the heap can happen [63, 99], which can lead to “unnecessary” memory allocations.

The shared libraries, which are used by the process, are typically stored between the heap and the stack segment [99].

Normally, the segments are ordered as described, starting from the lower addresses to the higher, but, this can differ in various architectures. The order can be seen in Linux-based operating systems (OS) using the process file system (procfs, see manual page of the procfs), e.g., in /proc/PID/maps.

The memory layout, as described above, applies to the virtual address space, also known as logical address space, of applications. The memory management unit (MMU) translates the virtual addresses to real addresses [99]. This translation is often assisted by the translation look-aside buffer (TLB) [116].

The main memory will be allocated by processes in pages, which are successive memory chunks with a size of “pagesize”[2] [99].

If a process allocates main memory, exceeding the available memory already fetched, a page fault will be created by the OS [99]. Two different types of page faults can happen: Minor- and major page fault.

A minor page fault, also known as soft page fault, happens if the newly allocated page has to be requested from the main memory. If the memory has to be fetched later from a higher level memory, e.g., hard disk drive (HDD), a major page fault, also known as hard page fault, is triggered by the OS. Minor page faults are less expensive in terms of the time than major page faults. The amount of page faults of a process can be read through the “getrusage()” function[3] or through the procfs, e.g., in /proc/PID/stat.

Memory Handling in Applications This section discusses several assets and drawbacks of the different memory allocation functions. More information about the explained function calls can be found in the corresponding manual pages.

[2]The pagesize can be evaluated in POSIX-based OS's using “sysconf(_SC_PAGESIZE)” from unistd.h

[3]The function can be accessed through sys/resource.h in Linux-based OS.

Initializing Memory Memory, which is allocated, will usually be filled with data. The "memset()" function can be used to initialize the requested memory to a predefined value. The time needed for initialization significantly depends on the amount of memory.

Allocating Stack Memory Usually, the stack is not handled by the programmer in a direct way. It serves as a highly dynamically memory for storing temporal used data. Nevertheless, the stack can be allocated, e.g., using the "alloca()" function. The implementation is very fast on most systems, as it is only adjusting the stack pointer register. As described in Section 3.3.2, the stack cannot be fragmented. As drawback, an allocation failure is not indicated and, therefore, it is often handled as the "out-of-stack" space situation, e.g., with a segmentation fault. Moreover, it is recommended to avoid the "alloca()" function with large unbounded allocations.

Allocating Heap Memory The heap memory is designed to provide a flexible run-time storage to the programmer. Whenever heap memory has been allocated, it has to be freed, e.g., using the "free()" function call, to avoid memory leaks. Further problems using the heap memory can exist, e.g., dangling pointers or freeing the same memory twice [84].

Common heap allocation functions are: "malloc()", "realloc()" and "memalign()". "Malloc()"[4] fetches main memory in multiples of system page sizes but uses only the requested memory. The remaining heap memory can then be used later on. The requested system pages do not have to be continuous in the main memory. Freeing the allocated memory can lead to heap fragmentation as described in Section 3.3.2. "Realloc()" is similar to "malloc()" but can be used to resize the memory as requested by the programmer. Moreover, "realloc()" acts like malloc if a NULL-pointer is given and can be used as "free()" if the "size" parameter is omitted. "Memalign()" basically uses the "malloc()" function and then aligns the obtained value. Moreover, the memory returned by "malloc()" is usually aligned, anyway. Thus, the use of the "memalign()" function is deprecated and only mentioned for the sake of completeness.

Another possibility to allocate heap memory is to use the "calloc()" function. "Calloc()" is designed to allocate memory for arrays. Here, the number of elements as well as the size per element can be specified. Additionally, "calloc()" initializes it is allocated elements to zero.

[4]Many different "malloc()" implementations exist, e.g., "dlmalloc()" or "ptmalloc()" (see [72]).

The available heap memory is usually handled by an ordered list [63]. Hence, the "malloc()" function call can be expensive if the heap is highly fragmented. Moreover, the heap has to be reconfigured if memory is newly allocated or freed by the process. Thus, using the heap memory is usually more expensive regarding the execution time than using the stack memory. However, memory from the heap can be used throughout the process run time and, usually, more heap memory can be allocated than stack memory.

Another aspect of the heap memory is that "freed" heap memory is often not directly returned to the system. Instead, it is stored in a malloc pool for further usage [113, 135].

3.4 Data Cache Memory Stubs

Most designers of architectures try to avoid an access of the main memory because of the time needed to load data from there to the CPU, e.g., about 240 cycles for an Intel Pentium M processor [28]. Therefore, the developers of CPUs implement several strategies of using caches to decrease the need to access the main memory and to increase the speed of execution, e.g., instruction- and data-prefetching [98, 132] or branch predictions.

3.4.1 Related Work

Cache behavior predictions for applications have been done in [33]. Here, the content of the caches is determined using an abstract semantics of machine programs.

In [79] an approach for creating data cache hits and misses has been described. The algorithm constantly access data and evaluates the amount of cache hits and misses by evaluating the timing behavior of the application. It aims at measuring the caching architecture of a system and provides information about the cache levels, cache line size, associativity and access times. The tool, as described in [80] determines information about data or unified caches as well as translation lookaside buffer (TLB) caches.

The approach used in [79] differs from our approach in the way that it constantly accesses data and the hit or miss rate is evaluated by timing measurements afterwards. In contrast to that, we want to simulate a desired amount of cache hits and misses, e.g., 1000 level two misses, 20 level two hits and 5000 level one hits, in a deterministic way.

3.4.2 Basic Literature

The main areas of research have been summarized in Figure 3.2. This section introduces these areas.

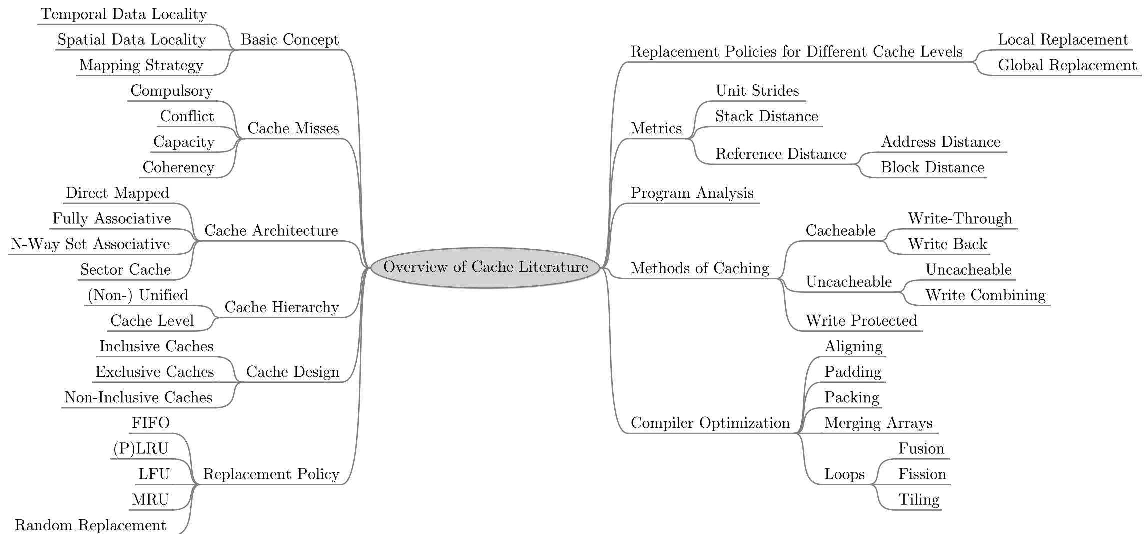


Figure 3.2: Literature Review on CPU Caches

Memory caches are buffers that are used to reduce the access times to the main memory. Therefore, they store information for reuse. As the access time to the caches are smaller (see Figure 3.1), the overall execution time of an application can be reduced. The reuse ability of data for caching is based on two assumptions: temporal- and spatial data locality.

Temporal Data Locality The first, temporal data locality is based on the idea that data, which have recently been referenced, will be needed soon and therefore should be stored in the cache. Many of the cache replacement policies, e.g., least recently used (LRU), are based on this assumption.

Spatial Data Locality The second is spatial data locality. The idea is that data that are declared related to each other in the source code, and therefore will be stored close to each other in the main memory, will be used at the same time in the application. This assumption is satisfied by always replacing a dedicated amount of bytes in the cache, which is called cache line, with successive data blocks from the main memory.

As the sizes of caches are smaller than the size of the main memory only parts of the memory can be stored. Therefore, many blocks of the main memory have to

be stored to the same location in the cache. The main memory address of data or instructions is normally used to determine which cache line the data or instruction will be mapped on the cache. The identification of the cache line, which will be replaced, e.g., because of capacity needs, is done by using "colored bits" [71, 98] and is often realized using the modulo operation.

Cache Misses A cache miss occurs if data will be referenced which are not available and therefore has to be loaded. In [7] a classification of different cache misses is described as the "Three Cs": compulsory, conflict and capacity.

Compulsory Misses Compulsory misses, also often referred as "cold" misses, can be seen if no data has been referenced to the specific cache line. Therefore, they only occur at the very first access to the cache line. These misses normally do not strongly influence the systems behavior [70], except in the initial phase.

Conflict Misses These misses occur if a still valid cache line will be replaced by a cache fill (see [51]) despite there are empty or non-valid data in the cache. These misses depend on the cache structure as discussed below.

Capacity Misses The last category capacity misses can be seen if a valid cache line will be replaced because neither an empty nor a non-valid cache line can be replaced.

Additionally, there is a fourth category of cache misses: coherency. These misses can only occur in multi-processing environments as they happen when data in the cache has been modified by another processing unit.

Cache Architecture The overall cache structure has been explained in [28, 39, 46, 132]. The cache structure highly affects the amount of different cache misses, e.g., there are no conflict misses in fully associative caches. There are three different types of the organization of a cache: direct mapped (DM), fully associated (FA) and n-way set associative (SA).

Direct Mapped In direct mapped caches every data or instruction will be mapped onto a dedicated cache line. The replacement algorithm normally uses the modulo operator. This approach can easily be realized. The drawback is that direct mapped caches can be "trashed" by constantly accessing data with a stride that is determined by the modulo operation of the cache size.

Fully Associative Despite of the direct mapped caches, in fully associative caches every data or instruction can be stored in any cache line. In this architecture there are no conflict misses as only misses occur if no as free tagged cache lines are available in the cache. Therefore all misses are either compulsory- or capacity misses. The main disadvantage of this cache architecture is the high complexity to determine the cache line that will be replaced if a miss happens. In the worst case every cache line has to be checked, which is an expensive operation.

N-Way Set Associative A combination of the direct mapped and fully associative cache is the n-way set associative cache architecture. In this caches “N” direct mapped caches are used in parallel. Therefore, “N” different cache lines are combined to a cache set. These cache sets can be seen as fully associative caches with “N” different cache lines. The number of different cache sets is the number of different cache lines in one of the direct mapped caches. A cache line fill will overwrite a cache line of the cache set which is determined using the modulo operator as explained in direct mapped caches. The cache line that will be overwritten in the cache set is determined as in fully associative caches and depends on the replacement policy as described below.

Sector Caches In sector caches each cache line is called sector [95]. These sectors consist of two or more subsectors. Each subsector has its own “valid” and “dirty” bits to determine whether the data stored in the subsector is valid or not. Thus, a sector can be partly filled.

A sector cache is organized as described in the “Cache Architecture”-Section (see above), e.g., as a n-way set associative cache. Hence, each data, which have to be stored in a subsector, will be mapped to the cache as in any other other cache architecture. To determine the cache set resp. cache line (i.e., sector), where the data will be stored, a “tag” is used [95]. This tag is part of the memory’s physical address. The data will be stored in a specific subsector of this sector. The subsector is determined by direct mapping.

In sector caches usually the whole sector will be fetched from the main memory or the upper cache level. However, the possibility exists to disable the sector fetching [18]. If the sector fetching is disabled and a cache miss occurs two possible cases have to be differentiated (see [52, 95]). First, if the data, which will be fetched, have a different tag to the tag of the data stored in the sector the sector is cleaned.

Now, the data are fetched and stored in the subsector, and, the tag is stored for this sector. In the other case, i.e., the data, which will be fetched have the same tag as the data in the sector, only the subsector has to be filled. The other subsectors of this sector are not changed.

Sector caches were designed to reduce the trade-offs of direct mapped caches, e.g., little data locality. With the successor, n-way set associative caches, cache designer temporarily stopped using the sector cache design. As the number of different sets in the cache increased and, therefore, the overhead for this design also statically increased the sector cache design has been revisited in [95]. Some modern central processing units include caches, which are designed as n-way set associative but incorporate the techniques of sector caches, e.g., the unified level three cache in Intel Xeon processors [51].

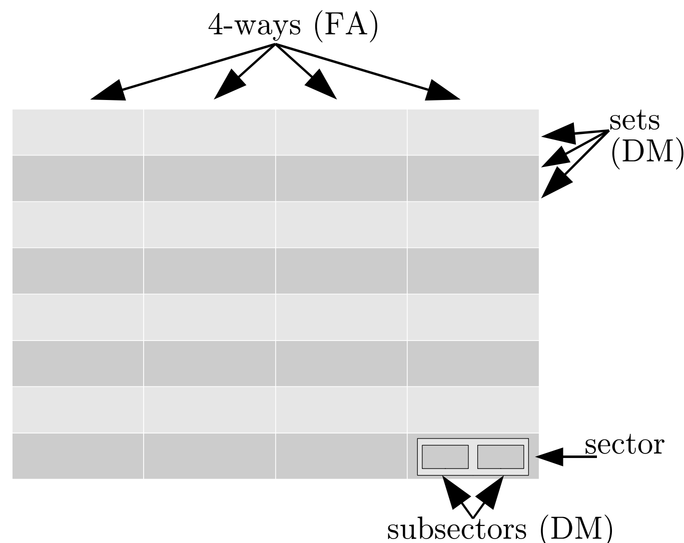


Figure 3.3: Example: 4-Way Set Associative Sector Cache

Figure 3.3 sketches the design of a “4-way set associative sector” cache. To evaluate the subsector where the referenced data will be stored to the caching algorithm firstly identifies the cache set realized as in direct mapped caches. Having the cache set, the replacement algorithm, as described below determines the sector which is done as in fully associative caches. The final place in the sector will be identified using the address (direct mapped).

A “4-way set associative sector” cache with two subsectors per sector is similar to an 8-way set associative cache. The overhead for determining the subsector is less than determining the cache line in a non-sector cache. The drawback of the sector caches is that often more conflict misses happen.

Cache Hierarchy In modern CPUs normally different caches are used for storing information separately: either instructions or data or micro-ops.[5] These caches are defined as non-unified. Additionally, there are unified caches that store data as well as instructions.

These different types of caches are often integrated into a cache hierarchy. It consists of different caches, which are arranged into levels. The number of the level depends on their logical distance to the CPU in terms of accessing times. The first level cache is next to the CPU that means accessing this level takes less time than accessing the level two cache (see [39]).

Each cache level can contain several caches. E.g., the first level cache can consist of a data- as well as an instruction cache.

Cache Design In cache hierarchies several different issues arise with the data organization in the caches. The problem here is the implementation to maintain identical data stored in the different cache levels. Based on this problem, three different cache designs have been build: inclusive, exclusive and non-inclusive caches [54].

Inclusive Caches In inclusive caches the inclusion property dictates that every data stored in a closer cache level has to exist in all other further cache levels. In this case the coherence mechanism is simply as the coherence scheme only has to probe the furthest cache level to invalid or update the cache line. If the data is not found in that level the data cannot exists in any other level. The drawback of this property is that the information has to be stored in all cache levels. Therefore, it is wasteful of space and bandwidth [139]. Additionally, as the information has to exist in all levels, the cache line size has to be the same all over the hierarchy.

Exclusive Caches Exclusive caches are the opposite of inclusive caches. In this design the information stored in a cache level are not in stored in any other level. In this case more data can be stored in the cache as compared to the inclusive cache design. As drawback, in this caching architecture every level of the cache has to be probed by the coherence mechanism instead of only probing the lowest level [54]. A performance evaluation of the exclusive cache design has been done in [141]. These caches can often be found in AMD CPUs.

[5] Micro-ops are stored in a so called trace cache, e.g., used in many Intel Pentium 4®.

Non-Inclusive Caches As a combination of the cache designs mentioned above, non-inclusive caches exists. This cache design is sometimes also referred as "mainly inclusive". Here, the designers attempt to include the advantages of exclusive and inclusive caches. Therefore, neither the exclusion nor the inclusion property is maintained [54]. These caches can often be found in Intel CPUs.

A comparison of inclusive and exclusive cache design regarding the worst case execution time has been done in [59].

Replacement Policies The hardware architecture uses replacement policies to determine the cache line that will be overwritten if a conflict exists. Different policies are explained in [3, 93]. [51] describes the policy used in some Intel Architectures. In direct mapped caches data are always stored to a specific cache line, there is no special replacement policy used. Within fully associative caches any cache line from the cache can be overwritten by any data from the main memory. Therefore, the replacement policy strongly influences the caching behavior. Determining the next cache line which will be overwritten causes a noticeable overhead to the execution time. For n-way set associative caches the replacement policy determines which cache line from the cache set has to be used.

The following replacement policies are discussed in this section (see [3, 93]):

First In First Out (FIFO) The replacement policy FIFO always overwrites the cache line which longest stay in the cache. It can simply be seen as a queue of a given length. An implementation of FIFO is often realized by a round-robin counter which is incremented after new data is loaded into a set [93].

Least Recently Used ((P)LRU) The least recently used (LRU) policy is the most popular implementation in computer caches [43]. It always overwrites the cache line which has not been used for the longest time and is often realized via a counter, which is incremented after the data is accessed [93].

The tree-based Pseudo-LRU (PLRU) as described in [93] approximates the least recently used data. This is realized by using tree-bits to point to the (approximated) oldest data stored in the cache set. The advantage of PLRU regarding the LRU is that the PLRU uses fewer bits to identify the next cache line. The drawback of this replacement algorithm is that it is only an approximation. This replacement algorithm is used in many different CPUs, e.g., in PowerPC 75x and Intel Pentium II, III and IV [93].

Least Frequently Used (LFU) The least frequently used replacement algorithm is similar to LRU. However, in LRU the oldest entry will be replaced in LFU the entry which has been used fewest.

Most Recently Used (MRU) The opposite of the LFU is MRU. In this case, the entry which has been used mostly will be replaced. The policy is based on the assumption that data which have often been used in the past will not be used anymore in the future. A nice explanation of the MRU replacement policy can be found in [93]

Random Replacement Random replacement strategies, e.g., implemented via a linear feedback shift register (LFSR) [3] choose the cache line which will be replaced randomly [3].

In [138] different cache replacement policies have been revisited. Additionally, different experimental studies have been done by using the least recently used policy as example.

Replacement Policies for Different Cache Levels In [138] the feasibility of global replacement policies is discussed. Therefore, the advantages and disadvantages of local replacement policies as well as global policies have been evaluated and supported by measurements.

Local Replacement In the local replacement scheme every cache keeps track of it is cache lines within each cache set to determine which cache line has to be replaced next. In this scheme every cache miss on a dedicated cache level influences the replacement policy of the next higher cache as a cache hit or miss occurs on the higher level. Therefore, a communication between different cache levels is established if a cache miss happens.

Global Replacement In addition to the local replacement, global replacement policies control the replacements of cache lines from a cache set in all hierarchies. There are two different types of global replacement policies: communication-based and centralized.

Communication-Based In a communication-based replacement policy, each cache level has it is own replacement policy. However, information regarding replacement will be exchanged.

Centralized Centralized replacement policy, is realized by a single controller which makes decisions for all cache hierarchies.

More information about global and local policies can be found in [138].

Metrics Several metrics to evaluate data locality and memory reference are discussed below.

Unit Strides Unit strides according to memory usage means, accessing data in the memory in strides with the same distance in terms of bytes. In [136] unit strides have been used to study the caching optimizations for scientific programs. Whereas, it has been considered as very efficient to use "unit stride" array accesses, some strides may cause cache trashing [136].

Stack Distance The stack distance, as proposed in [7], measures the distance in time between two references of the same data location and quantifies temporal locality. This reuse metric is applied to stack distance with cache line granularity instead of data granularity.

Reference Distance Another metric is called reference distance. It has been proposed as a metric for data locality and is the total number of access to the same block of data [7, 89]. The reference distance has been split into "address distance" and "block distance" in [89]. The "address distance" is defined with a block size of one and is a metric for measuring temporal locality. If the block size is greater than one, the "address distance" is called "block distance" and is targeting on spatial locality.

Program Analysis In [33] program analysis by abstract interpretation of caching behavior has been presented and applied to predict the cache behavior of applications for real time systems. There, the memory references have been split into the categories "always hit", "always miss" and "not classified". These categories have been used for two analyses: "must" and "may". The "must" analysis determines if a set of memory references is definitely in the cache if the program flow reaches a given program point. The "may" analysis is used to determine if a memory block may be in the cache and is finally used to guarantee the absence of the memory block of the cache.

Methods of Caching Normally, system memory is split into several regions. As theoretical every region of the system memory can be cached in the processor [51], these regions are tagged to be handled according to their usage. Therefore, different strategies exist to handle modified data stored in a cache [39]. According to [51] a main classification into cacheable and uncacheable can be done.

Cacheable In this category data from this region of the system memory can be cached. To write back the modified data into the system memory different policies exist: write-through and write back.

Write-Through (WT) The write-through policy updates the system memory whenever data has been modified. Therefore, the system memory holds all changes to the data. This causes overhead if the same data will be overwritten frequently but not read from the system memory. In this case, a single update of the system memory would be sufficient at the last data access.

Write Back (WB) To reduce the overhead of the write-through policy, the write back accumulates writes to cache lines and forwards the updates to the system memory when a write-back operation is performed. This reduces the bus traffic but requires snoop operations (see [54]) to the system bus to ensure the memory and cache coherency.

Uncacheable There are three different types to set data from the main memory to uncacheable:

Strong Uncacheable (UC) To prevent data from caching memory regions can be set to strong uncacheable (UC). In this case, all references have to be satisfied by the system memory. This behavior is useful for memory-mapped I/O devices.

Uncacheable (UC-) The same behavior as for strong uncacheable (UC) applies to uncacheable (UC-). However, these system memory regions can be explicitly set to cacheable by changing several CPU register settings.

Write Combining (WC) In write combining the system memory is not cached and the bus coherency protocol does not enforce coherency (see [44]). To reduce system bus traffic, writes are combined in the write combining buffer (WC buffer) and maybe delayed before the data will be written to the system memory.

Write Protected (WP) Write protected memory types are a combination of cacheable and uncacheable memory. In this case, data can be cached for read events. However, write events cannot be stored in the cache. In this case, writes are propagated to the system bus.

The above mentioned memory types strongly depend on the system architecture and may not be available in every system.

Types of Caches There are several different types of caches in a modern CPU. The types are separated by the kind of data, which can be stored (see [15, 17, 61]). The most important types are described in the following.

Unified Caches An unified cache is often used in the higher levels of the caching architecture and stores instructions and data simultaneously. Intel CPUs are using unified caches since 1993 [28]. These caches are typically read- and writable [17].

Data Caches These caches are used to store data [28]. They are available in many CPUs such as the Intel Pentium 4 [17] on the lower cache levels. These caches are usually read- and writable, too [17].

Instruction/Trace Caches Instruction and trace caches are used to store the code of a process. An instruction cache stores the instructions of the application and a trace cache is used to store decoded instructions, called µops. They are mainly used in the lower levels of the cache architecture [28]. These caches are typically read-only, beside that write events can occur, e.g., by using self-modifying code. Usually, either a trace or an instruction cache exist in the CPU. More information about trace caches can be found in [15, 47, 54].

Translation Look-aside Buffer Memory addresses used in the binary of an application are usually virtual, i.e., these addresses must be translated into physical addresses for being able to access the information stored in the main memory. This translation is typically costly in terms of time. To avoid recursive translation of a memory addresses a buffer, called translation look-aside buffer (TLB), has been added to the cache architecture. These TLBs can exist to store data (dTLB) or instruction (iTLB) memory addresses. More information can be found in Section 3.3.2.

Write/Store Buffers Write buffers store data, which have been modified and not updated in the higher levels of the cache or main memory [17]. Write buffers, which are used together with an out-of-order execution are often referred to store buffers [120]. More information about write buffers can be found in [12].

Victim Cache/Buffer A victim cache (see [57]) stores data, which have been evicted from the cache. The data are stocked in the cache in case they are referenced again. Hence, conflict misses can easily be avoided without adding more associativity to the cache. These caches are typically of small size and fully associative. A victim cache is often referred as a victim buffer [59].

Load Buffer These caches are often used together with the out-of-order execution logic of modern CPUs. They store information to perform the parallel instruction execution, e.g., the decoded instructions and address information. More details about the out-of-order execution logic are described below.

Out-Of-Order Execution The out-of-order execution logic of a CPU enables the CPU to reorder multiple instructions of the program's code. There are several dependencies on whether the instructions can be reordered or not (see [15, 29]). E.g., an Intel Pentium 4 CPU can only reorder read instructions. Write instructions, however, are always executed in the order they appear in the application [15]. To improve the out-of-order capabilities of the CPU store and load buffers are often used.

Compiler Optimization Many compiler optimization techniques are targeting to improve the data locality and data spatiality [89]. Therefore, they try to move data which will be referenced successively close to each other. Many optimization techniques regarding memory are described in [71]:

Aligning Aligning strategies have been developed to increase data locality and to possibility pack data, which are used together in the application [70, 98]. They aim to store data instead of two different cache lines into a single cache line and therefore reduce the capacity or conflict misses.

Padding In [94] *inter-variable* and *intra-variable padding* are described. These are array transformations to avoid conflicting distance between uniformly-generated references. It can be realized by linearizing array references and

calculating the distance. To improve the caching behavior padding variables will be added to the arrays to avoid trashing in direct mapped caches.

A similar padding strategy is described in [70]. Additionally, the opposite of padding is described as *packing*. This optimization technique packs arrays into the smallest possible place in order to increase data locality.

Another approach is called *merging arrays*. This is similar to packing but targets to combine two (or more) different arrays of the same dimension with the same indexes. The result will be a compound array to increase spatial locality.

Loops In [7, 70] *loop fusion* is described as a technique to increase instruction level parallelism and therefore optimize the distance of references. In this case, the same data will be accessed more often and therefore will not be replaced by other data.

Loop fission This is the opposite to loop fusion. In this case the program transformation split portions of the loop body in different loops [70]. The same optimization is described in [7] as “loop distribution”.

Loop Tiling As described in [7, 89], loop tiling tries to keep the amount of referenced data between the use and reuse smaller than the cache size. This improves the possibility that the data can be found in the cache and therefore do not have to be fetched from the main memory or the next higher cache level [70].

Another possible improvement technique is called *loop interchange* [89]. It targets to interchange loops for fully permutable loops, e.g., in matrix multiplications.

3.5 Simulated Software Functionality

A key functionality of the *dynamic performance stubs* is to record and to recreate functional behavior using the *simulated software functionality*. The recording requires the serialization of internal data-structures into a format from which they can be recovered at a later point. This functionality has been predominantly implemented in distributed systems where objects and code are marshaled for exchange between peers.

3.5.1 Related Work

Most closely related to our serialization approach is the work in [117, 118] in which a “MPI Serializer” has been introduced. The target of this project is the efficiently and automated marshaling of C++ data structures. The tool generates automatically marshaling and unmarshaling code for the message parsing interface (MPI), which is often used as communication interface in high performance computing (HPC). The “MPI Serializer” is is based on the C++ serialization possibility of the “GCC-XML” project [37], which uses the gcc abstract semantic graph (ASG) scheme [69] to determine the serialization specification.

To some extent, our approach is similar to [117, 118] as both projects needs to serialize C++ data structures. But, it differs in many details. E.g., it has been decided to store and restore the functional behavior of software modules, which will be replaced by a stub. This can be used to remove a software bottleneck. In contrast, the focus in [117, 118] is to provide marshaling code for the message parsing interface.

However, both projects are based on the abstract semantic graph scheme provided by the “GCC-XML” project.

In [13], a lightweight fact extractor is presented. It utilizes XML tools, i.e., XPATH and XSLT, to extract static information from the C++ source code files. The approach is to transfer the source code into “srcML”, which is a XML representation of the file. The fact extractor is mainly used to parse and search the source code. This technique is often used for reverse engineering, maintenance, testing or even in general development of software systems. This approach is based on “CPPX” [20], which is an open source C++ fact extractor. The fact base, which is generated by “CPPX”, can be used as input for software development tools, such as integrated development environments (IDE). It enhances these tools’ functionalities, for example by source code visualization, object recovery, restructuring and refactoring.

As in [117, 118], the approach of [13] highly differs from our approach, as it is not supposed to store and recreate the functional behavior of software modules. [13] mainly delivers a XML presentation of the extracted facts of the source code.

3.5.2 Basic Literature

In order to access each member of C++ data structures or classes for the *simulated software functionality* an internal representation of the data structure is necessary.

This can be achieved by serializing the data structures or by using the "abstract sematic graph". Moreover, a technique to gain access to the members of the serialized objects has to be found. This section discusses the three mentioned areas.

Serialization of Objects

Serialization is a concept, which can be used to convert a class into a binary stream containing all members of the class including the members of subclasses. This can be used to store and/or restore the state of the object during the runtime. Moreover, the serialized objects can be simply stored for further analyses or used to be sent over the network. This is often done by using communication stubs (see [14]), which uses the (un-)marshaling concept (see [19]).

The basic mechanism is to serialize the objects into a flat binary representation of the object or to read the flat representation and to recreate the object.

Many programming language support the concept of serialization within the language definition, e.g., in Java classes can implement the "Serializable" interface for getting access to the serialization of the objects [30]. The C++ programming language does not support the serialization concept. Here, several serialization libraries exist to support the developer, e.g., the serialization library [91] (see [24, 62]) of the boost C++ libraries project [21].

Abstract Semantic Graph

The abstract semantic graph (ASG) represents the semantics of an expression in programming languages. It extends the abstract syntax tree (AST) by including additional information such as type information [90].

As this ASG presents the internal structure of the application, it can be used to evaluate several dependencies of the data structures or classes. Hence, the ASG can be used as input for serializing the data structures. This is done in [20], which extends the "GCC" as described above.

Access to Members of Classes

A possibility to directly access private or protected members of C++ classes from outside of the class is to use the "friend" mechanism, e.g., "friend functions" and/or "friend classes" [27, 114]. In this case, the full prototypes of the external functions or classes, which should be able to access the members have be declared as "friend"s

within the source class. Here, the functions or classes, which are set as “friend”s are not considered as members of the source class.

However, the “friend”ship mechanisms is usually deprecated in an object-oriented programming software design as it potentially violates the data encapsulation paradigm. E.g., it should be considered whether the functionality of the external friend function can be directly included into the source class itself.

Chapter 4

Dynamic Performance Stubs

This chapter describes the dynamic performance stubs framework. Additionally, it shortly provides the main contributions, which will be described in this thesis, e.g., the performance simulation functions or the simulated software functionality.

4.1 Basic Design Decisions

The major decisions regarding the design and implementation of the *dynamic performance stub* (DPS) are:

1. The programming language used in the project is C/C++. Therefore, the *DPS* have to be written in C. This language has several advantages such as the possibility of inline assembler code if a high optimization level is necessary. Additionally C/C++ can be compiled and used as a binary executable code for the native machine language of the CPU used. This is highly recommended in [1] since the programming language influences the performance of the complete system. Moreover, most of the C/C++ compilers provide different optimization flags, e.g., [38]. However, the methodology of the *dynamic performance stub* is independent to the implementation language and can also be used with interpreted, just-in-time compilation or intermediate code.

2. The simulation functions should be configurable and adjustable to the target. This should be done only once during the setup of the stubs.

3. The simulation functions of the performance behavior should be used as a toolset. All functions should be accessible without further need of configuration.

4.2 Concept

The concept of *DPS* combines the methods of software testing [6, 73, 109] and performance improvements [55, 35, 74, 42]. There are only little differences between stubbing for performance improvements and stubbing for testing reasons. Normally, stubs are used for simulating remote systems or for non-existing software modules and functions, as in software testing [41]. In this approach the CUS will be replaced by *DPS* in order to simulate the performance behavior of this software unit as a primary goal. This procedure relates to stubbing a single software unit and, hence, it will be called “local”. The *performance simulation functions* (PSF) can also be used to change the behavior of the complete system. Therefore, a software module has to be created which interacts “global” in the sense of influencing the whole system instead of only one software unit.

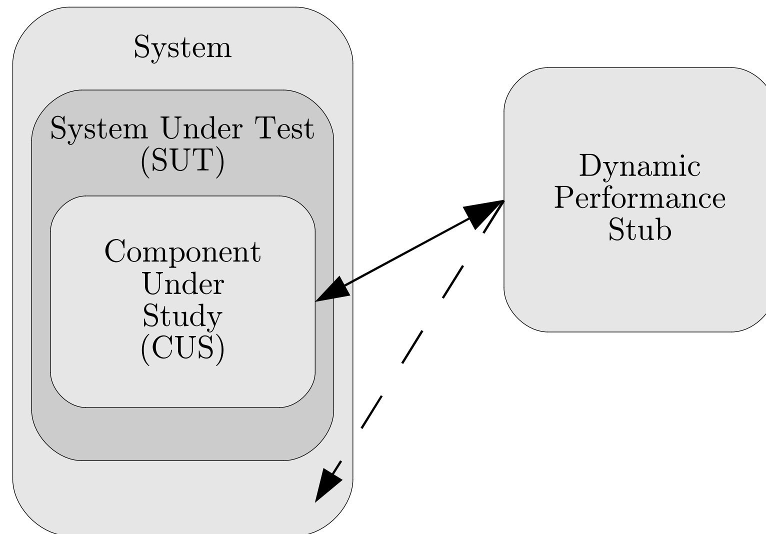


Figure 4.1: Interactions of *Dynamic Performance Stubs*

This is presented in Figure 4.1. Here, the line with two arrowheads depicts the replacement of the component under study (CUS) by the *DPS*. In this case, the *DPS* is a *local DPS*. The dashed line shows an extension of the SUT by a *global DPS*.

4.3 Framework

A *dynamic performance stub* (DPS) consists of two major components, which are: *performance simulation functions* (PSF) and the *simulated software functionality* (SSF). The performance behavior of the stub can be dynamically adjusted to the needs of the performance simulation study, e.g., after each performance test run.

The name "*DPS*" generally means that a stub is used, where the performance behavior can be changed. The stub that will be used is typically a realization of the *DPS*. The name of the stub reflects the main component, which will be simulated and adjusted during the performance measurements, e.g., a *main memory stubs* is used to simulate the main memory behavior of the application by using the *main memory PSF*. Whereas the execution time of the CUS is also simulated, the main focus is on the main memory behavior. A not closer specified realization of a *DPS* is referred as *particular dynamic performance stub* (PDPS).

The framework of the *dynamic performance stub* consists of the following parts, which are presented in Figure 4.2:

- Simulated Software Functionality (SSF)

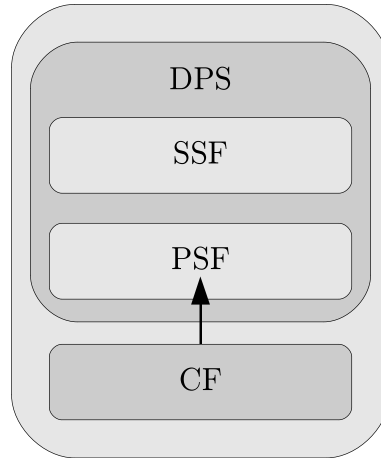


Figure 4.2: Framework of the *Dynamic Performance Stubs*

- Performance Simulation Functions (PSF)
- Calibration Functions (CF)

The *SSF* is additionally implemented code in order to simulate the functional behavior of the already exiting CUS. More information about a possible methodology of stubbing for already existing software modules will be described in Section 4.5.

After creating the functional stub (using the *SSF*) the performance behavior of the CUS can be modeled using the *PSF*. They will provide the possibility to simulate different isolated performance parameters such as the time spent in the component. The functions can be combined in order to simulate the "real" performance behavior. Please refer to Section 4.4 for an overview of the performance parameters, which can be simulated by the *PSF*.

Additionally, the framework of the *DPS* contains the *calibration functions* (CF). These are important for the initial setup of the *PSF* to the target, e.g., the time needed for an "empty loop" will be determined in order to setup a realistic behavior. The *CF* have to be executed once for each hardware system.

4.4 Performance Simulation Functions

A *PSF* simulates the non-functional behavior of a function regarding to one aspect of performance. They are divided into several classes according to the performance bounds to which a software can belong.

An overview of the several *PSF* realizations is depicted in Figure 4.3. The

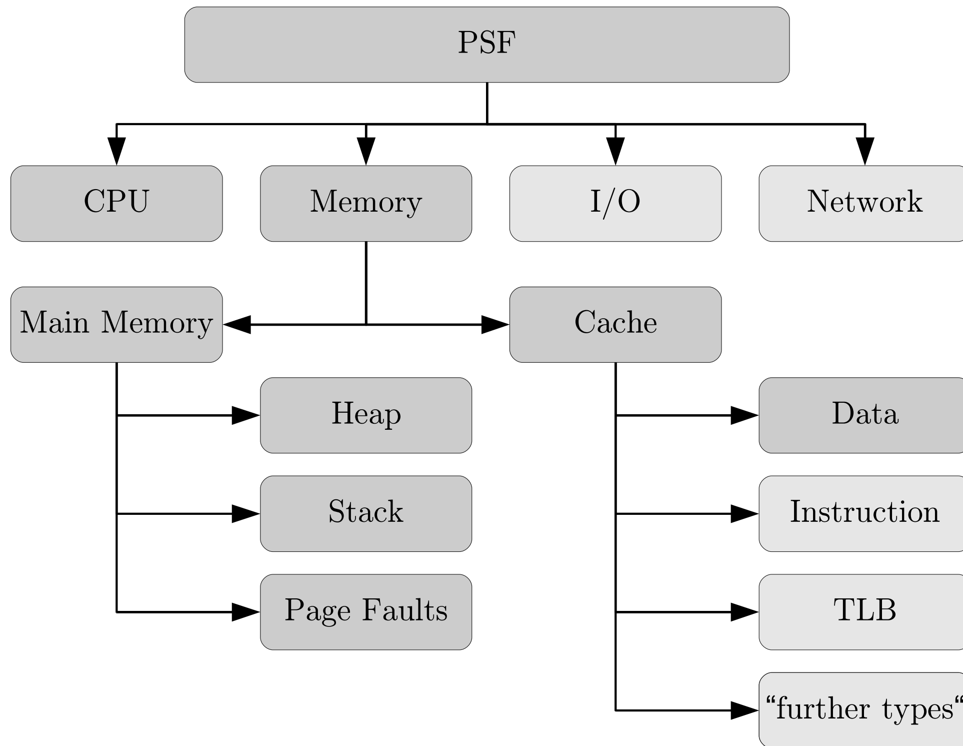


Figure 4.3: Classification of *Performance Simulation Functions*

dark-colored boxes presents the *PSF*, which will be handled within this thesis. The light-colored boxes can be seen as future work.

The main classes are as follows:

- CPU
- Memory
- I/O
- Network

For a more detailed description on the main classes the reader is referred to [48].

A fifth category can be thought of is “user” [55], in our approach it will be included for simplification in the I/O category of the *PSF*. A closer look in the categories will be given in the following subsections.

4.4.1 CPU PSF

This *PSF* basically relates to the CPU states, which a process can have and, hence, to the clock cycles used by the process. One part of the *CPU PSF* is the time spent

in the stub, while it is scheduled. The other part is the time, while the process is blocked. Here, the CPU can be used for further processes.

4.4.2 Memory PSF

Memory with respect to *PSF* refers to the volatile storage. More in detail the different types of memory described in this sections are: "main memory" and "cache memory".

Main Memory PSF

Main memory PSF refers to the dynamically allocatable memory segments of a process, i.e., "heap" and "stack". Moreover, page faults, which are caused while the execution of the software modules or functions will also be simulated within their memory segments.

Cache Memory PSF

Cache memory PSF simulates the access behavior of the software module or function regarding to the caches. The main categories of caches are:

- Data Cache Memory PSF
- Instruction Cache Memory PSF
- Translation Look-aside Buffer PSF

Within this thesis, the *data cache memory PSF* are examined. The other cache types as well as further caches types, such as, "store/write cache" or "victim buffer", will not be handled.

4.4.3 I/O PSF

I/O in computer science describes the way to handle the input and output values. Regarding to performance improvement of user applications I/O can be described as the interface between the application and the kernel realized by specific systemcalls [71, 75]. During the execution of these calls the process is blocked due to "I/O" and will be scheduled out, which means, the CPU will be freed from the process until the call has been finished and the process will be scheduled in again [71]. Systemcalls cover all interactions between the kernel and the user application this includes

also the memory and the network. However, these parts are already discussed in other sections of the *PSF* and will not be handled here. Hence, only the secondary memory, e.g., hard disks are taken into account. Further and more detailed studies in this area of *I/O PSF* are necessary.

Those *PSF* will not be handled within this thesis.

4.4.4 Network PSF

The *network PSF* will handle possible performance parameters specific to the network. Especially, the following network metrics will be considered to simulate the network behavior for various load situations:

- One-Way Delay
- Round-Trip Time
- Delay Variation
- Packet Loss
- Packet Reordering

For more details of the network metrics the reader is referred the according requests for comments (RFC) [4, 5, 23, 56, 68, 85, 88, 112].

Those *PSF* will not be handled within this thesis. A follow-up research project to evaluate *network PSF* has already been started.

4.4.5 Calibration Functions

Some different values have to be set up in order to provide the suggested simulation properties. This can be done using the *CF*. The idea behind these functions is to execute the *PSF* with several different input values and to trace the according output results. The calculation of the desired values will also be done inside of the *CF*. Using the results will provide a proper setup for the usage of different *PSF* in order to stub the CUS. The *CF* are providing additional functionality, such as they will report if something unexpected was happening. In this scenario, they will also give hints on how to improve the measurement for a proper calibration. As an example: a context switch happened while the configuration of the *CPU PSF*. The *CF*' trace includes a warning that a context switch was happening. Additionally, it will give a hint like raising the priority of the *DPS* in order to get valid results.

4.5 Simulated Software Functionality

> *"...the simplest way to manage the call is to build a stub, that is, a procedure that has the same I/O parameters as the missing procedure, but a highly simplified behavior. For example, the stub might produce its expected results by reading them from a file or requesting them from a human tester interactively; or it might even do nothing and simply print some diagnostic message, should be acceptable to the caller. The stub will then be linked with the module, just as if it were the real procedure."*
> [41]

Commonly a stub in software engineering is used as a proxy and provides an adequate replacement for the behavior of the to-be-implemented software modules and functions. It is mainly used in software testing of distributed systems or in modularized software [109].

Thus, the stubbed function uses the same I/O parameters to simulate the missing procedure and provides only a basic functionality, e.g., it returns the results by reading them from a hash table, does nothing or it may simply write trace messages. From the systems' point of view the stub will be included and work just as the real procedure [41].

The idea behind the *SSF* and the functional stubs is nearly the same. The only difference between stubbing for performance reasons and the generation of "test-stubs", as mentioned, is the part of software, which will be used. Stubbing for performance reasons means to replace an already existing code, whereas, stubbing for testing creates basic functionality of non-existing software [41, 109] or they are used for remote systems.

A methodology of stubbing already existing and deterministic software functions can be described as follows. First of all, the in- and output values of the CUS have to be identified while tracing them. Therefore, proper tracepoints have to be inserted in the source code and, then, the software has to be re-run with a proper and deterministic test case scenario. Rewrite the CUS using the in- and output values (e.g., using a hash table) and by replacing the time consuming functions where the software normally walks through. Now the stub should be working properly.

4.6 Definition of a General Methodology

As a fundamental prerequisite a reproducible automated test of the system has to be available. The system's use has to be restricted to the task of performance optimization. Without such a test procedure, it is not possible to reproduce the performance results and, thus, to validate the results or even more to isolate performance bottlenecks. In addition, the *calibration function* has to be parametrized according the used hardware platform as sketched in Section 4.4.5.

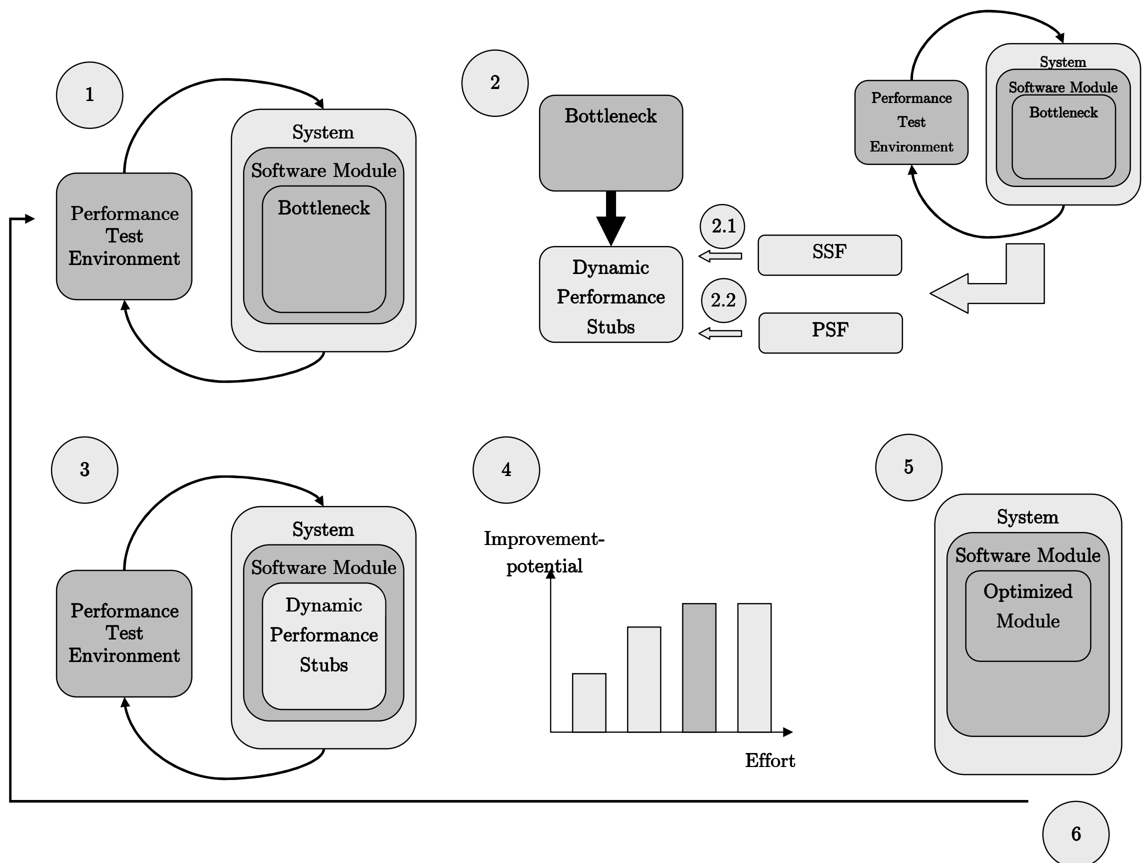


Figure 4.4: Methodology for Using *Dynamic Performance Stubs*

The method for performance optimizations using *DPS* is depicted in Figure 4.6 and described by the following steps:

1. *Identification of performance bottlenecks.*
   As a first step, some possible bottlenecks of the origin software have to be identified. First a performance measurement of the whole SUT has to be realized. The delivered performance data has to be interpreted and some possible

bottlenecks have to be identified. This can be achieved, if all performance data are available in a very detailed level. So a "drill-down" method on the delivered performance data can be used. Please note that the success of this step strongly depends on the experience and the in depth knowledge of the software of the analyzing person. As a result of this step, a list with possible performance bottlenecks can be generated. For further information, it is referred to [55, 74].

Now, a first cost-benefit analysis of optimizations should be done. Please note that this ratio only might be estimated. The result of this step is a prioritized list of bottleneck candidates.

2. *Now a dynamic performance stub has to be generated.*
   The CUS has to be studied. This also includes a necessary level of abstraction, especially, which functions have to be stubbed. A *DPS* can now be generated using the following steps:

   2.1 The functional behavior of the CUS has to be simulated. This will be done by the realization of the *SSF*.

   2.2 The performance behavior of the CUS has to be determined and simulated. The simulation will be realized by the *PSF*.

   The correct function of the *SSF* and the *PSF* have to be validated. So performance measurements have to be done with the stubbed functionality. The results should be in the same range as the original performance results. If this is not the case, the *PSF* has to be modified. This might also happen if the analysis for this bottleneck candidate is not sound and as a consequence it has to be reconsidered.

   As an additional check a performance measurement with only a *SSF* can be done and the resulting data should be analyzed very carefully. In addition the next bottleneck might be visible.

3. *Several measurements changing the performance behavior of the stub should be realized.*
   The performance data has to be analyzed carefully. This can be done as described in Step 1. Additionally, several different charts can be drawn. They will probably indicate different system behaviors such as a changeover from CPU to memory bound or a changeover in the critical path in the SUT.

4. *Cost-benefit analysis.*

   A cost-benefit analysis using the possible gain can be applied and the effort for improving the performance bottleneck can be estimated based on the evaluation data. As a further result, there might appear new bottlenecks, which should be checked using the method in Step 1. If new bottlenecks appeared, this might be also a hint to further similar bottlenecks elsewhere is the system, e.g., deep copies can also happen at further system ports.

   Based on this data the candidates for spending optimization efforts can be chosen and the work on improvement can be started.

5. *Verify optimization gain.*

   The optimized components should be included in the software and a new performance measurement should be started.

   The achieved data should be compared with simulated data. If there is a huge discrepancy the method for finding bottlenecks or even *PSF* should be corrected.

6. *Verify the systems' performance.*

   If the software has still not the desired performance, goto Step 1.

4.7 Extensions to the Methodology

In many cases the aforementioned method can be used. However, it can be seen that performance analysis is a highly sophisticated and specialized task. In the following section, some possible techniques or alternatives to the method described before have been sketched. These can be seen as some ideas, which might help on the overall task like a partially filled toolbox.

4.7.1 Mixture of PSF

Sometimes an isolated performance behavior is not sufficient for simulation. E.g., using only the *CPU PSF* might lead to false results, if the component uses in addition to a possible heavy usage of the CPU also disk I/O very often. This is the case especially in bigger CUS. A solution is to combine several different types of *PSF* in order to yield a more accurate simulation result. However, the degree of combination might be analyzed very careful. This is especially on bigger CUS no easy task.

4.7.2 Full and Partial Stubs

Another problem appears on bigger CUS, too. The components behavior cannot be simulated in a realistic way by a simple stub, where a collection of input values delivers an output value. Usually, it might be an arbitrary interleaved mixture of calling subroutines, internal calculations and delivering results. This can be realized by extending the stub with additional *SSF*, which is calling the real procedures and discarding the results from it or storing them for further analysis, e.g., in a hash table.

The same technique can be used for functions which could not be optimized and therefore is no need to be simulated.

4.7.3 Idealized Measurements

The stubs can also be used to simulate almost ideal performance behavior of the CUS. This is useful for a feasibility study or for verifying the specified performance targets of third party software, e.g., middleware and can be realized by only using the capabilities of the *SSF* without inserting other functions, such as *PSF*. This will tear down the cycles to a minimum and, hence, the "other" software modules can use almost every resources in total. Of course, the operating system will also contribute it is cost to the total load.

4.7.4 Load and Stress Tests

Another possibility is the opposite to the idealized measurements. Here, the system will be stressed by adding additional performance bottlenecks to evaluate the system behavior under a high load. It can be realized by using a *global dynamic performance stub*, e.g., an application, which is executing the *CPU PSF* combined with a high priority. This will increase the CPU load and, thus, will stress the system. Depending on the software, overload routines will be performed in such situations. So the behavior of the application on the borderline can be examined and new bottlenecks can be identified.

4.7.5 Hidden Bottlenecks Detection by Zero Bound CUS

Hidden bottlenecks are bottlenecks, which are existing but can only be seen after removing the current one. They are also reducing the throughput but it does not

count to the overall performance due to the stronger influence of the first bottleneck. Specialists often "over optimize" the software module, which leads to the effect that the hidden bottleneck becomes critical [84].

The *DPS* help in detecting hidden bottlenecks. Here, the CUS will be optimized in a non realistic way by setting the performance values of the *PSF* to zero. So, a new performance measurement will help to make the next bottlenecks visible.

4.7.6 System Bounds

If all local optimizations do not lead to the required results or are too expensive, sometimes a global optimization might be chosen, for instance using a faster CPU or a bigger memory. To check whether global restrictions are there *DPS* can help. E.g., after evaluating several runs with different performance parameters a chart can be drawn. This can point to a system restriction, e.g., the system starts swapping/-paging due to a lack of memory.

This can also be used to estimate the performance of future systems, especially, if a ramp-down of the system resources are planned.

4.7.7 Global vs. Local Stubs

As described in Section 4.2, there are two different occurrences of *DPS*: *local-* and *global DPS*.

First, they can work *locally* which means to stub a software unit. This procedure is handled along the thesis and will not be described more in detail in this section.

The second way of using the *DPS* is *globally*. Thus, a software application (module) has to be written, which will affect the whole system in a desired way, e.g., by generating CPU load.

As an example: a module will be created, which compares the current usage of the CPU with a default value. If the current value is less than the default, the module will use the CPU, e.g., with the *CPU PSF* and tries to adjust the usage by generating load. Otherwise, it will not do anything. As a result global considerations can be realized. It can be checked whether the throughput of the whole system is still on a limit. E.g., if after a linear increase of the CPU load by a global stub lead to a non-linear increase of the runtime of the process to be examined. This might be an indication that the system throughput is too low to handle all requests. Then, an increase of the CPU power might help. However, this might not be an option in all projects, but also then, this examination result helps in gaining a

better understanding of the systems behavior and in consequence of the possible performance problem.

Another possibility for the usage of a *global dynamic performance stub* is to increase the number of context switches. For this, a software module can be created, which only wakes up and goes back to sleep again. If it will only sleep for a short time and is executed with a high priority the system will do a context switch. This module can be realized using the *CPU PSF*.

Moreover, *global DPS* can be used as a benchmark suite, which is highly customizable to the behavior of the real application. Here, several different *global DPS* can be created, e.g., several *CPU stubs* and *memory stubs*. These stubs are executed on different systems and their performance measurement results are compared. This compares the systems against each other for the dedicated workload, which is based on the real application.

4.8 Advantages

As mentioned, the *dynamic performance stub* can be used for a cost-benefit analysis. This will also lead to a balance between optimization effort and the achievable gain in the system. Of course, a higher optimization of the software module will also lead to a higher performance within the complete system, but, the effort for the additional gain in this case might be to big. This approach can lead to a more gain-oriented optimization. This point will end up in better maintainable and structured code. General drawbacks of performance improvements, such as a poor maintainability and badly structured code, are described in [1, 87]. These drawbacks can be reduced by the presented approach.

Additionally, knowing the optimization effort can lead to results earlier, because some possible improvements does not have to be done due to system dependencies [84].

As described in Section 4.7.4, “load & stress” tests are possible. This can also be used for testing functions, e.g., overload routines under “real” conditions because the CPU utilization will be raised up by a *global dynamic performance stub* module. As mentioned in Section 4.7.3, a similar methodology can also be used for idealized measurements regarding the performance behavior of the CUS.

The *dynamic performance stub* can be used to identify hidden bottlenecks as described in 4.7.5.

4.9 Restrictions

The *DPS* have also some drawbacks, which will be described in this section. The first to mention is the danger of wrong results of the initial performance measurements. If this error happens all further results of the measurements using the stub can also be wrong. This can lead to a lot of time wasted in measuring, building and evaluating the performance. As mentioned in [55] all results should be handled with care until they are validated.

Also with proper initial measurements and a proper stub setup, the results can be misleading, e.g., because of the introduced overhead. Hence, as already stated, all results should be handled with care.

Creating a *dynamic performance stub* means effort, which can require additional costs. The gain of this method strongly depends on the system, its performance behavior and the effort to be spent for optimizations.

Large projects often take a lot of additional effort for performance stubbing, each iteration of stubbing one element requires a change of parameters and the repetition of the build- and performance measurement process as well as the evaluation.

Within the software life cycle the interfaces and the messages of the software can change. This means probably that a stub has to be adapted after each of this changes.

There is a lot of additional effort as result of measurement operations. However, using the described method should decrease the overall improvement effort especially on large software systems.

4.10 Summary

Using *DPS* following advantages for performance improvements can be achieved:

- The results of possible performance optimizations can be estimated with an increased confidence level, because a complete program execution with a simulated performance optimization can be achieved before the optimization has been realized. As a consequence, a validated estimation of a performance improvement can be given before the effort of a concrete optimization has been spent. This optimization effort can sometimes be several man months instead of the effort for generating a performance stub which in most cases can be measured in hours. Since the benefit of performance improvement can now be

determined, a valid cost-benefit analysis for improvement operations can be calculated, if the effort for each operation has been estimated. So different candidates for performance improvements can be prioritized and such the effort of improvement can be spent on a section with a big return on investment.

- Using *DPS* also the necessary performance gain can be determined. Sometimes an improvement of, e.g., the reduction of the execution time of a bottleneck to 50% does lead to the same execution time for the whole application as a reduction to 75%, because other components will be new bottlenecks. So by *DPS*, the necessary ratio of improvement operations can be determined and in consequence an overengineered performance improvement can be avoided.

- Also the "hill climbing phenomena" can be avoided. Usually, on mountains hiking only the next peak can be seen in advance and later higher hilltops on the way are hidden. This usually happens with performance bottlenecks too. If you have reduced one, the next bottleneck appears [84]. Here *DPS* can be used to see the upcoming bottlenecks in advance.

The simulation possibilities of the *DPS* are subdivided in several *PSF*. The following sections cover the simulation behavior of the CPU, main memory and data cache memory.

Chapter 5

CPU Stubs

The CPU performance simulation functions can be used to model the time behavior of software modules or functions. This chapter discusses the requirements, a possible realization as well as a methodology to use these. Moreover, the calibration functions for the CPU performance simulation functions are described.

5.1 Requirements

The *CPU stubs* should be able to remodel the performance behavior of a CUS regarding the time. Hence, the following two different states of a process shall be modeled: “working” and “blocked”. The first presents the process while it is executed by the CPU. The second is used to model the time periods while the process is scheduled out.

Moreover, *CPU stubs* should be able to simulate the time periods with high-precision for short- and long periods. Moreover, *CPU stubs* should be able to use a dedicated amount of the CPU, e.g., to constantly use 5%. The algorithm is based on an “open loop” in this case. Additionally, they should be able to regulate the CPU utilization to a predefined value, i.e., the algorithm evaluates the CPU utilization and adjust its own CPU utilization accordingly (“closed loop”). Of course, the CPU utilization can only be increased.

The approach of the *CPU PSF* shall be portable to other architectures and programming languages. Hence, the *CPU PSF* shall be executable on different platforms by only modifying some values, which has been determined by the *calibration functions* (CF).

5.2 Realization of the CPU Performance Simulation Functions

CPU stubs are used to simulate the CPU usage behavior of software modules or functions in order to simulate a CPU bound software bottleneck. A CPU can either wait or can be active. If the CPU is in the wait state the “idle” process will be executed. Otherwise, i.e., the CPU is active, another process in the system is executed. From a process perspective, a process can be either executed by the CPU or can be blocked. Hence, the states, which will be simulated by a *CPU stub*, can be defined. Those are: “system influencing” and “system non-influencing”.

System Influencing: This state simulates a process, which is currently executed. Hence, the process uses the CPU. From a CPU perspective, this can be seen as “running” according to the process states of [116]. This part of the *CPU PSF* is called “system influencing” as the CPU has to execute instructions. These can be either from the *CPU stub* or from another process. Hence, the CPU has to be busy.

System Non-influencing: This state simulates a process, which execution is currently delayed by any reason, e.g., because the process waits for user input or because is the CPU is not available. This states depicts the process states "blocked" or "ready" (see [116]). This part of the *CPU PSF* is called "system non-influencing" as the process does not influence the system. Here, the CPU can be either idle or can execute another process.

An example implementation of a *system influencing* and *system non-influencing* realization of the *PSF* simulating the CPU is given below.

Example for a System Influencing CPU PSF The *system influencing* stubbing of CPU cycles can be realized using "no operation" (NOP) so that only the resource CPU will be used by the *PSF*; but, other system components are not. They have to be executed in a loop in order to reach the desired time consumption. Additionally, the NOPs and also the loop have to be protected against compiler optimizations.

Using this method has some advantages and restrictions. On the one hand, it is easy to implement and calibrate. Moreover, it is mainly architecture independent and only slightly modifies other parts of the system. Almost all needed time values can be simulated by this implementation. They can range from some nano seconds to several minutes or more (see Section 9.2.1).

On the other hand, this so called "busy loop", can lead to undefined results if the CPU highly uses the CPU frequency scaling possibility. In contrast to the "waiting" time, the duration needed for processing a single cycle depends on the actual CPU frequency. If the CPU uses the frequency scaling feature the ratio between "waiting" and "working" will not be constant. Therefore, errors can be introduced in the test results if the frequency of the CPU is changing. However, the time to execute one cycles is not constant in any system, which utilizes the CPU frequency scaling feature. Hence, the time needed to execute a cpu bound process varies even in these systems. So, this behavior can be expected to be "normal" and only has to be considered for the test results evaluation.

Implementation An example implementation can be seen in Listing 5.1. The function "useCycles" takes a value standing for the processor cycles working for "usec" as input. The "TIME" constant is defined within the *DPS* and has been evaluated using the *CF*. Here each iteration will exactly consume 1μs while looping

around the empty statement ";". The same approach is taken inside of the Linux kernel. Here, the "BogoMIPS" [8, 75] value will be evaluated while booting. The result is stored in the "loops_per_jiffies" variable and used for small delays, e.g., within the linux kernel ndelay function.

```
void useCycles (long usec)
{
    long i;
    for (i = 0; i < usec * TIME; i++)
    {
        ;
    }
}
```

Listing 5.1: Example Implementation of a *System Influencing CPU PSF*

Example for a System Non-influencing CPU PSF The simulation of the *system non-influencing CPU PSF*, which means that the real process is blocked or waiting for an event, can be handled easily by letting the *CPU stub* sleep for the desired time. Of course, it cannot be guarantied that the *CPU stub* is promptly executed after the sleep period in non real-time systems. But, there is no difference between the execution of a *CPU stubs* and any other process. Hence, this behavior is regarded as normal and does not influence the usability of the *system non-influencing CPU PSF*.

Implementation For instance in UNIX environments the usleep()-function can be used. For details see "unistd.h".

By using the *system influencing* and *system non-influencing* elements of the *CPU PSF* a *CPU stub* can be created, which simulates the CPU behavior of a software function or, even, of a whole process.

5.3 Calibration Functions

The *CF* are used to adjust the *CPU PSF* to a dedicated hardware architecture. Several steps have to be taken for setting up the *CPU PSF* to the system. In common, the setup consists of mainly two parts: "calibration" and "validation".

Calibration of System Influencing CPU PSF For the *system influencing CPU PSF*, the number of loops for a predetermined duration has to be evaluated, e.g., the value of the $TIME$ constant in Listing 5.1. This can be done as shown in Listing 5.2. The calibrateLoop()-function has to be called, varying the "number of loops" value, until it returns the desired time value with the needed precision.

```
1  long long int calibrateLoop (long nLoops)
2  {
3      long i;
4      long long int beforeTSC;
5      beforeTSC=readTSC();
6      for (i = 0; i < nLoops; i++)
7      {
8          ;
9      }
10     return ( readTSC() - beforeTSC );
11 }
```

Listing 5.2: Example Implementation of a *System Influencing CPU CF*

The readTSC()-function basically executes some assembler instructions to read the value of the time stamp counter (TSC), which is a hardware register [2, 31]. This value is updated with each processor cycle. Of course, the overhead for the readTSC()-function call has been determined and subtracted. A similar approach has also been taken to calculate the "bogoMIPS" value[1].

In order to ensure the quality of the result the execution of the function shall not be interrupted. This can be achieved by the following preconditions:

- Executed with high priority
- Executed on an idle system
- Using a tickless kernel
- Adding a short "system recovery" time between each execution. So, other processes can use the CPU without disturbing the *CF*.

Additionally, it has to be ensured that no interrupt took place, e.g., using the getrusage()-function call.

[1] see Linux kernel source init/calibrate.c

Validation of the Results After determining the "number of loops", the result has to be validated. The first step is to validate the proper working for the predefined amount of time by executing the calibrateLoop()-function statistically sufficiently often with "number of loops". If the results are sound, the second evaluation can take place. Here, the proper working of the *system influencing CPU PSF* for the time range, which is necessary to simulate the timing behavior of the bottleneck, has to be validated. Hence, the "number of loops" has to be recalculated using the rule of proportion and the measurements have to be done again. To improve the accuracy of the results, the steps as described above can be taken. The quality of the results can be evaluated by applying the linear regression method to the measured values and calculating a confidence interval.

Calibration of System Non-Influencing CPU PSF The *system non-influencing CPU PSF*, as described in Section 5.2, are only using the system internal waiting functions. Because of, these functions are already delivered by the operating system, nothing has to be adjusted here.

5.4 Methodology

In this section, we present a methodology of using *CPU stubs* to evaluate the maximum improvement factor, which can be achieved by optimizing a CPU bound software bottleneck. With the parallelization of processing tasks by modern architectures and operating systems, concurrency issues and analysis of individual CPU usage becomes increasingly important. This methodology can especially be used within in multi-processing or multi-core systems settings to evaluate concurrency issues.

1. *Determination of the CPU bottleneck:*
   The SUT has to be defined and a suspected bottleneck (CUS) has to be identified, which is done by common software performance engineering (SPE)[104, 106], e.g., profiling or tracing. Now, several performance indicators have to be determined:

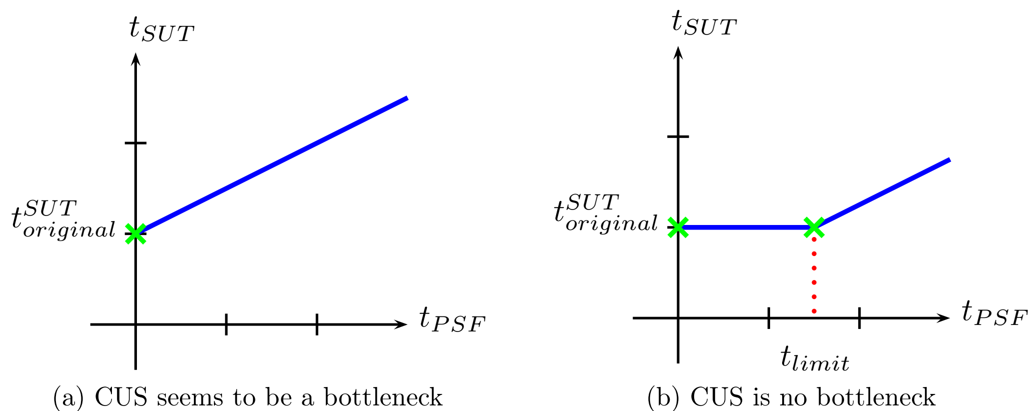
   - t^{CUS}: Time spent in the bottleneck (CUS).
   - t^{SUT}: Time spent in the software module or function (SUT) from which the CUS is part of.

- t^{CUS}_{busy}: Time spent in the CUS using the CPU. It includes the user-mode time as well as the system-mode time (see [116]).
- $t^{CUS}_{waiting}$: Time spent in the CUS waiting to be scheduled (see: process state "Ready" in [116]).
- $t^{CUS}_{blocked}$: Time spent in the CUS waiting for an event (see: process state "Blocked" in [116]).

The measured values have to be deterministic within several performance test runs.

2. *Validate CPU bottleneck:*

   In this step, a simple validation of the chosen CUS will be done. Thus, the *system influencing CPU PSF* is inserted in front of the CUS and the performance measurements will be repeated increasing the time spent in the *PSF* (t_{PSF}). The measured time of the SUT mainly follows one of the diagrams given in Figure 5.1.



(a) CUS seems to be a bottleneck

(b) CUS is no bottleneck

Figure 5.1: Validate Component Under Study as a Bottleneck

   In Figure 5.1a the increase of the *system influencing CPU PSF* leads to an arithmetically increasing amount of time spent in the SUT. Therefore, the CUS seems to be a CPU bottleneck. Hence, the next step can processed.

   In the other case, Figure 5.1b shows that an increase in the execution time of the CUS does not increase the time spent in the SUT for $t_{PSF} < t_{limit}$. This points out that the CUS is no bottleneck for the system. Another potential CPU bottleneck has to be identified (Step 1).

   This step can be done to remove overhead as it excludes the CUS from being mistaken as a CPU bottleneck easily. This step is optional.

3. *Study the bottleneck performance behavior:*
   The value $t^{CUS}_{blocked}$, which has been determined in Step 1, will be used to evaluate the CPU utilization of the CUS.

   A value of $t^{CUS}_{blocked} = 0$ means that there are no waiting periods triggered by the CUS while executing. So, the process will not be interrupted by the CPU except there are external events, e.g., scheduling. In this case, the methodology can be used as provided.

   A value of $t^{CUS}_{blocked} > 0$ means that the process switches to the "Blocked" state. Here, the trace files recorded in Step 1 have to be studied further, in order to identify successive working and waiting periods of the process. The following steps of this methodology have to be done for every working period starting from the biggest to the smallest working period. The waiting period will be simulated with the *system non-influencing CPU PSF*. The simulated waiting time will normally be constant, if no further reduction caused by optimizations of this time period can be expected.

4. *Flat CPU Stub - Evaluate the optimization potential:*
   Now, a *flat CPU stub* will be used to determine the optimization potential. A *flat CPU stub* is a *particular dynamic performance stub* (PDPS), which only simulates the functional behavior of the CUS using the *SSF*. Hence, it only introduces small overhead in the system and can be used to simulate the ideal time behavior of the CUS. This can be used to analyze the maximum performance gain of the SUT as it is not the same as $t^{CUS} = 0$, especially in multi-core or parallel processing environments. As the final result often depends on several in parallel working threads or processes. Therefore, the following values have to be measured:

   - t^{STUB}_{flat}: Time spent in the *flat CPU stub*.
   - t^{SUT}_{flat}: Time spent in the SUT including the *flat CPU stub*.

   An indicator of the possible optimization amount can be evaluated by calculating:

   - $t^{CUS}_{reduced} = t^{CUS} - t^{STUB}_{flat}$: The time, which has been reduced in the CUS.
   - $t^{SUT}_{reduced} = t^{SUT} - t^{SUT}_{flat}$: This describes the total possible optimization gain.

If the CUS is executed more than once in sequence, the maximum number of iterations per CPU (*iter*) has to be evaluated. Hence, $t^{CUS}_{reduced} * iter$ and $t^{SUT}_{reduced}$ has to be compared. It is possible to use the factor *iter* because the same CUS is executed, so each iteration takes the same amount of time. The values t^{CUS} and t^{SUT} are taken from Step 1. Now, the calculated values can be compared and the following cases can be evaluated:

- $t^{CUS}_{reduced} = t^{SUT}_{reduced}$: This means that, the SUT directly depends on the CUS. Hence, there are no system dependencies, i.e., "hidden bottlenecks". Additionally, no "over optimization", as described in [84], can be done. The more time optimized in the CUS the better it is. In this case, the next step of this methodology is Step 7, i.e, optimize as much as possible. However, in case of an expected hardware bottleneck, Step 5 can be done. This behavior is typically for batch or procedural processing in single core environments.
- $t^{CUS}_{reduced} > t^{SUT}_{reduced}$: In this case, the possible optimization amount is less than the time spent in the CUS. Thus, there are system dependencies, which have to be studied further and we can move on to the next step. This behavior can mainly be seen in multi-core and parallel processing systems. As there might be parallel threads or processes, which additionally delays the execution after the actual bottleneck has been reduced. This is particularly the case if a change over in the critical path has happened (see [11]).

The case $t^{CUS}_{reduced} < t^{SUT}_{reduced}$ does not has to be considered. This would mean that the speed up of the execution time in the SUT is more than has been reduced in the CUS. Hence, the execution time of $t^{SUT} - t^{CUS}$ would has been decreased, but, the software within this part of the SUT has not been changed.

As it is only an indicator, the time $t^{SUT}_{reduced}$ delivers no information about the amount of optimization, which has to be done in the CUS, especially for $t^{CUS}_{reduced} > t^{SUT}_{reduced}$.

5. *Idle CPU Stub - Evaluate system dependencies:*
   Here, the *flat CPU stub* will be extended using the *system non-influencing CPU PSF*. This is called an *idle CPU stub*. The total simulated time is the total processing time of the CUS (t^{CUS}_{busy}). Hence, the following equation holds $t^{CUS}_{busy} = t^{STUB}_{idle}$. Where, t^{STUB}_{idle} is the time spent in the *idle CPU stub*. Now,

the performance measurements will be redone and the t^{SUT}_{idle} value, which is the total execution time of the SUT including the *idle CPU stub*, shall be recorded.

Dependencies between an *idle CPU stub* and the system can be evaluated using the values: t^{SUT}_{idle} and t^{SUT}. Thus, the total execution time of the original SUT will be compared to the execution time of the SUT using the *idle CPU stub*. The following cases can be separated:

- $t^{SUT}_{idle} = t^{SUT}$: This means that the total execution time of the SUT has not changed due to the usage of the *idle CPU stub*. Whereas, the *idle CPU stub* only uses the CPU at the very first beginning and than hands the CPU over to the system. However, the total execution time of the SUT has not been changed. Hence, it can be concluded that no other process is blocked by the CPU. Therefore, adding CPUs to the system does not provide a significant performance improvement. Nevertheless, as of Step 2, the CUS is the bottleneck.
- $t^{SUT}_{idle} < t^{SUT}$: Here, the total execution time of the SUT decreases by using an *idle CPU stub*. Therefore, further processes are at least partially available in the "Ready" queue. In this case, these processes can be executed earlier. Therefore, the total execution time decreases. An optimization of the CUS as well as an additional CPU would decrease the total execution time.

The case that $t^{SUT}_{idle} > t^{SUT}$ does not have to be considered as it means that reducing the amount of instructions would lead to a longer execution time. This is not possible in typical CPU bound systems.

This step evaluates dependencies between running processes in the system and the CUS. Moreover, information about the influence of adding CPUs to the system can be achieved. However, the measurements do not provide any information whether a faster CPU will increase the total execution time. Albeit expected that a faster CPU will increase the total execution time. As the process is CPU bound, the amount of instructions determines the total execution time. Using a faster CPU means that each cycles and, hence, an instruction, is executed faster.

6. *Busy CPU Stub - Cost estimation:*
The *flat CPU stub* will be extended with the *system influencing CPU PSF*.

Now, the performance measurements will be repeated and the time spent in the *system influencing CPU PSF* (t_{PSF}) will be varied from zero to the total execution time of the CUS (t_{busy}^{CUS}). Typically, the time spent in the PSF will be increased by 10% of the total execution time for each iteration. This can also be redone if a particalur time slice, e.g., between 20% and 30%, identified a change over in the critical path as explained in this step. The following values have to be measured:

- t_{busy}^{STUB}: Time spent in the *busy CPU stub*.
- t_{busy}^{SUT}: Time spent in the SUT including the *busy CPU stub*.

Using these results, two different types of bottlenecks can be distinguished:

(a) **Total bottleneck:**
In this case, the measured values of the execution time from the SUT is linearly increasing. Thus, an optimization of the CUS will always result in an improvement of the execution speed and, thus, decrease the latency. This result should have been already achieved in Step 4.

(b) **Limited bottleneck:**
If the processing of the SUT depends on other functions respectively on their results, the graph might look similarly as given in Figure 5.2. The graph is split in two parts. In the first part, $t_{PSF} \leq t_{limit}$, the time of the SUT is constant at a minimum value (t_{min}^{SUT}). Within this area, the chosen CUS is no bottleneck to the system as an increasing in the amount of processing (t_{PSF}) does not lead to an increased execution time (t_{SUT}). At t_{limit} the behavior of the CUS chances to a CPU bottleneck. As can be seen in the figure, the time spent in the SUT increases along the time spent in the *system influencing CPU PSF* (t_{PSF}). This evaluation shows that an optimization of the bottleneck can only decrease the latency in the SUT to a given value (t_{min}^{SUT}).

These information can be used to identify "hidden" bottlenecks, e.g., a "hidden" bottleneck appears at t_{limit} of Figure 5.2. This limit is basically the maximum, which can be achieved by an optimization of the CUS. Hence, it can be compared to a changeover in the critical path (see [11]). Additionally, the information can be used for a cost-benefit analysis. Thus, a gain-oriented improvement can be done.

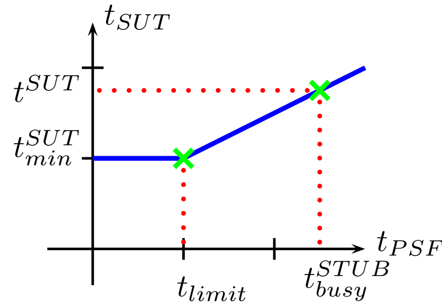


Figure 5.2: Limited Bottleneck

7. *Optimization of the software:*
   Now, the software module or function has to be optimized. Hence, the results from the cost-benefit analysis can be used for a software improvement related to the optimum between cost and effort. Finally, the performance of the software component has to be measured again. A new bottleneck has to be identified (first step) if the results show that the performance targets are not achieved yet.

The application of *CPU stubs* will be shown in the next section. Hence, the provided realization and methodology are applied to a case study.

5.5 Case Study

This sections evaluates the application of the *CPU stubs* to a performance optimization study on a step-by-step analyzes. First, the evaluation environment is described. This is followed by the description of the system under test. The last subsection describes the evaluation of the methodology.

5.5.1 Evaluation Environment

The application runs on a Linux operation system (Kernel 2.6.30.9). The evaluation is done using the Linux Trace Toolkit next generation (LTTng) [78]. For timing measurements, the TSC is used. The hardware is based on an Intel Centrino Core 2 Duo CPU. The calibration of the *CPU PSF* is realized as described in Section 5.3.

5.5.2 Original Function

The application used for the proof of concept is split into three highly CPU bound processing parts. The middle part calculates the results in two different threads and the last part needs both results to complete the calculation.

5.5.3 CPU Stubs

The described methodology has been applied to the before mentioned application. The measurements of each step have been done several times in order to evaluate some statistical behavior.

1. *Determination of the CPU bottleneck.*
   The SUT is defined as the whole application. The identified bottleneck is supposed to be in the parallel processing part of the application. The measurements have been repeated five times. Table 5.1 lists the average value for

	seconds (avg)	SCV
t^{SUT}	7.21	0.0000231
t^{CUS}	2.48	0.0002236
t^{CUS}_{busy}	2.42	0.0000130
$t^{CUS}_{waiting}$	0.06	0
$t^{CUS}_{blocked}$	0	0

Table 5.1: Determination of the Performance Behavior

   each parameter as well as the squared coefficient of variation (SCV) [111]. The $t^{CUS}_{waiting}$ has been calculated. Hence, the SCV is zero.

   In this step, the SUT and the CUS have been determined and some more detailed analysis has been done. For a definition of the values see Section 5.4 Step 1.

2. *Validate CPU bottleneck.*
   Here, the *system influencing CPU PSF* has been added to the CUS and the timing behavior of the SUT has been studied. The evaluation of the result has shown a similar behavior as presented in Figure 5.1a. Hence, the bottleneck has been validated.

3. *Study the bottleneck performance behavior.*
   The evaluation of $t^{CUS}_{blocked}$ has shown that the application is 100% CPU bound,

i.e., the process did not stall. Hence, no special steps are necessary in the following steps.

4. *Flat CPU Stub - Evaluate the optimization potential.*
   In this step, the *SSF* has to be built without any *PSF*. Afterwards, the time spent in the *flat CPU stub* (t^{STUB}_{flat}) has to be measured as well as the time spent in the system under test (t^{SUT}_{flat}). This measurement has been done five times in order to get some statistical distribution validation.

	seconds	SCV
t^{SUT}_{flat}	6.28	0.0000059
t^{STUB}_{flat}	0.000000034	0.0084648

Table 5.2: *Flat CPU Stubs* Timing Measurements

   The results have been summarized in Table 5.2. As can be seen, the time spent in the stub (t^{STUB}_{flat}) is approximately zero. The total execution time of the SUT (t^{SUT}_{flat}) is about 0.93 seconds, which has been defined as $t^{SUT}_{reduced}$, less than the SUT time measured in Step 1 (see Line 2 of Table 5.1). However, the time spent in the CUS ($t^{CUS}_{reduced}$) has been reduced by about 2.48 seconds (see t^{CUS} in Line 3 of Table 5.1). As $t^{CUS}_{reduced} > t^{SUT}_{reduced}$, this is an indicator that there are further influences of the system.

5. *Idle CPU Stub - Evaluate system dependencies.*
   The *flat CPU stub* is extended by the *system non-influencing CPU PSF*. This step basically evaluates whether an additional CPU might improve the total execution time. The measured average time of t^{SUT}_{idle} is 7.15. The comparison of this value with t^{SUT} from Step 1 pointed out that an additional CPU would not result in a significantly improved execution time of the SUT. This result can be easily explained, as there are two CPUs for executing two parallel processes and the rest of the system is idle.

6. *Busy CPU Stub - Cost estimation.*
   This step determines the amount of CPU time spent in the CUS, which should be reduced for an ideal optimization. Hence, the *system influencing CPU PSF* (t_{PSF}) will be successively increased starting from zero to t^{CUS}_{busy}. The values t^{STUB}_{busy} and t^{SUT}_{busy} have been measured.

   Figure 5.3 evaluates the t^{SUT}_{busy} and t^{STUB}_{busy} (y-axis) depending on the ratio t_{PSF}/t^{CUS}_{busy} (x-axis). Whereas, the value for t^{STUB}_{busy} is increasing arithmeti-

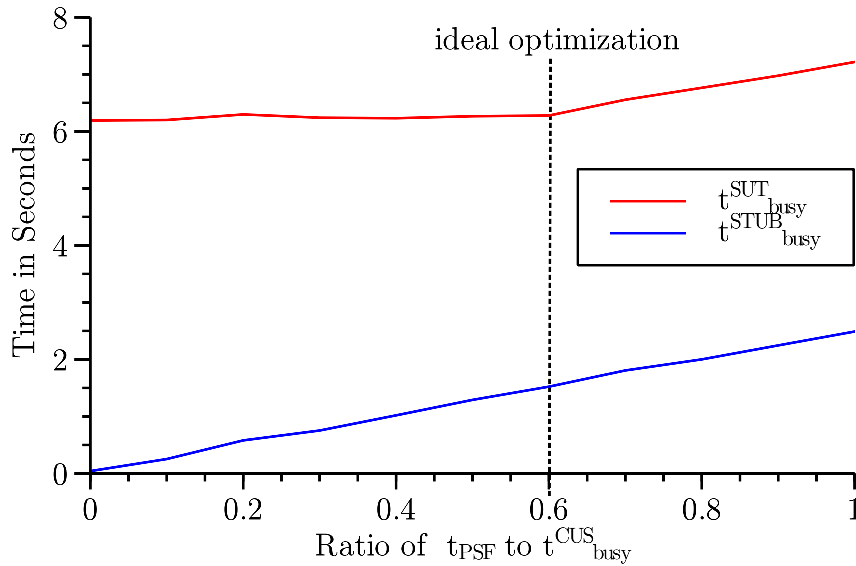


Figure 5.3: Optimization Effort Estimation

cally, the value for t^{SUT}_{busy} remains constant until 0.6 and than starts to increase arithmetically. Here, a changeover of the bottleneck can be seen. In this case, the CUS is only a bottleneck above 0.6 for the system. Hence, in order to ideally optimize, the time spent in the CUS has only be reduced to $ideal_optimization$ times t^{CUS}_{busy}. Now, a cost-benefit analysis can be done by an effort estimation of the optimization.

7. *Optimization of the software.*
   In this step, we highly "over optimized" the CUS to clearly identify the "hidden" bottleneck. We were able to reduce the time spent in the CUS to 0.280 micro seconds. However, the time spent in the SUT was still 6.29 seconds, which is close to the expected value as achieved in Step 4. Now, the next bottleneck should be optimized if necessary.

The presented methodology can be used to optimize parallel processing system. We have shown that, a possible optimization gain can be evaluated and "hidden" bottlenecks can easily be identified. This leads to a more gain-oriented optimization of the software.

5.6 Summary

In this chapter, we outlined the *CPU stubs*, which are a subset of the *DPS* framework. These stubs are used to simulate the CPU behavior of a software functions.

Thus, some requirements has been defined and the *CPU PSF* has been described as well as the *CF* are explained. A methodology to use *CPU stubs* within several systems, e.g., for multi-core systems, has been developed and evaluated by using a proof of concept.

The results clearly presents that by using *CPU stubs* it is possible to determine the potential outcome of a performance improvement study without actually improving the software module or function. This can be used for a cost-benefit analyzes and, hence, an over optimization can be avoided.

The next chapter describes the *main memory stubs* which are one realization of the *memory stubs*.

Chapter 6

Main Memory Stubs

The main memory performance simulation functions can be used to model the heap and stack behavior of software modules or functions depending on the time when the allocation will be done. This chapter discusses the requirements, a possible realization as well as a methodology to use these. Moreover, the calibration functions for the main memory performance simulation functions are described.

6.1 Requirements

To simulate the main memory behavior of the CUS accurately, the algorithm has to meet several requirements:

1. **Allocate heap or stack memory in chunks.**
   The allocation behavior of a software function usually allocates the memory in chunks. This behavior should be simulated in the *main memory stubs* (mm stubs), i.e., do not always (de-)allocate the total amount of memory.

2. **Allocate or free a defined amount of memory.**
   This requirement particularly relates to the stack memory segment as it is not possible to use a "free" function call to release the allocated memory. Here, the function, which has allocated the stack memory has to be terminated.

3. **Allocate or free memory at any given time.**
   It should be possible to change the allocated memory amount of heap and stack independently at each given sampling value.

4. **The allocated amount of memory can remain constant.**
   The amount of allocated heap and/or stack memory does not have to be changed at a given time.

5. **Use the allocated memory.**
   In order to create the necessary page faults, the allocated memory should be used.

6. **Create a page fault at their occurrences.**
   Recreate the page fault behavior of the software function as soon as a new main memory page is allocated.

By meeting the requirements, it is possible to simulate any main memory allocation behavior of software modules or functions. Additionally, the necessary page faults are created at the time when they are occurring.

6.2 Realization of the Main Memory Performance Simulation Functions

In order to meet the requirements of the algorithm the single conditions have to be regarded and proper solutions about how to fit them have to be found. The solutions presented in this section use the basic memory management functions that are provided from the GNU C Library.

(De-)allocate Heap Memory Heap memory is allocated by the dynamic use of memory within the SUT. This is usually done by calling functions like malloc(). To deallocate the heap memory, the free()-function is used. In the case of the simulation algorithm, using malloc() and free() to simulate the heap memory behavior would result in some problems. With the free()-function the total amount of heap memory allocated by malloc() would be freed. So a decrease of heap memory would be simulated by freeing the total amount of heap memory and allocating the requested amount afterwards. This would finally result in the correct amount of heap memory but it would not simulate the behavior in a correct way (see Requirements 1 & 4). The allocated amount of heap memory has to shrink but not to be freed and allocated again. For that reason, the presented algorithm uses the realloc()-function to allocate heap memory, which is able to change the size of the used heap memory in the desired way.

```
1 char* mHeap=NULL;
2 mHeap=(char*) realloc(mHeap, size);
3 if(mHeap==NULL) {
4   die(error);
5 }
```

Listing 6.1: *Heap Main Memory PSF*

A possible realization is presented in Listing 6.1. Here, the memory is allocated, reallocated and freed by using the realloc()-function call. The function will terminate the process if the requested amount of memory cannot be allocated.

(De-)allocate Stack Memory Stack memory is usually allocated during function calls when pushing the needed information, like the value of the base pointer, the return address and the local variables, onto the stack. The stack memory is freed by leaving the called function. To simulate this behavior a certain predefined amount

of stack has to be allocated when calling the simulation function. By leaving this function the allocated amount of stack will be freed. In order to rebuild several function calls, the simulation function has to be called recursively. Freeing the stack to a certain value, can only be done by leaving the recursion to a certain level. This leads to the requirement that this special amount of stack has to be simulated by a call of the simulation function.

By calling and leaving the simulation function in a recursive way, it's possible to simulate the rising and trailing edges of stack allocation. But, the call of the simulation function allocates a constant amount of stack. So, a possibility to allocate additional stack memory has to be found. For that the alloca()-function is used. It allocates the desired amount of stack memory by usually just moving the stack pointer and returning a pointer to the newly generated area of the stack memory. Thus, the behavior of the stack can be rebuilt (see Requirements 2 & 4).

```
1 char* mStack=NULL;
2 mStack=(char*) alloca(size);
3 // no error, if no space is available
4 // cannot be freed by a function call
```

Listing 6.2: *Stack Main Memory PSF*

In Listing 6.2, a possible realization for a *stack main memory PSF* (stack mm PSF) is presented. The amount of the stack is allocated by using the alloca()-function. This lines have to be included into the function, which shall modify the amount of stack memory. As of the alloca()-function, no error is indicated if the alloca()-function fails to allocate the requested amount of memory. Moreover, the function, which includes the alloca()-function has to be terminated to free the stack memory.

Simulate Time Delays In order to not only simulate the amounts of the SUTs heap and stack memory but also to simulate them at a correct time, the time delays between the changes of memory have to be considered. To simulate these time delays the usleep()-function, which is part of the unistd.h (libC) in Linux OSs, is used (see Requirement 3).

Using the Allocated Memory When executing the SUT the allocated memory is used. For that reason it is possible that page faults occur. The simulation algorithm recreates the memory behavior of the SUT, and so, it also has to rebuilt the

page faults that are produced within the original software. To invoke those page faults, the simulation also has to use the allocated memory (see Requirements 5 & 6). The "distmemset()-function" is introduced to generate the desired page faults with only small influences on the time behavior of the simulation.

```
1 void distmemset(char *memory, char init, int allocateSize, long
     int pagesize) {
2   int i = 0;
3   for (i = 0; i < allocateSize; i = i + pagesize) {
4     memory[i] = init;
5   }
6   if (allocateSize > 0) {
7     memory[allocateSize - 1] = init;
8   }
9 }
```

Listing 6.3: System Influencing Allocation of *Main Memory PSF*

In Listing 6.3, the distmemset()-function is presented. It will access each newly allocated page at least once to use the allocated memory. Hence, the requested amount of page faults are created. As one byte of the allocated page is used the execution time of the distmemset()-function is short. Similar behavior could have been achieved by using the memset()-function; but, the execution time is far higher.

In the next subsection a possible realization of an input file for the *mm stubs* is presented.

6.2.1 Simulation Data File

To reduce the overhead created when running the algorithm, the measuring points used to simulate are written into a data structure within a header file that is used to compile the algorithm.

Listing 6.4 shows an example header file that can be used to build the simulation algorithm. For each measuring point, the data structure "memAlloc" contains the time elapsed since the previous point. Additionally, the change of heap as well as stack memory are contained. Every point is presented by one "memAlloc" struct. Those structs are inserted into an array of "NUMDATA" length. So the algorithm can easily switch to the next trace point and does not produce much overhead in execution time.

```
#define NUMDATA 4

struct memAlloc{
  int time;
  int stackAlloc;
  int heapAlloc;
} memUse[NUMDATA]={
  [0].time=129, [0].stackAlloc=300, [0].heapAlloc=0,
  [1].time=223, [1].stackAlloc=100, [1].heapAlloc=200,
  [2].time=384, [2].stackAlloc=-100, [2].heapAlloc=-40,
  [3].time=112, [3].stackAlloc=-300, [3].heapAlloc=-160
}
```

Listing 6.4: Design of the Measuring Points' Data Structure

6.2.2 Algorithm

This section describes a realization of the *mm stubs* based on the requirements (see Section 6.1) and *mm PSF* (see Section 6.2). The algorithm uses the *simulation data* (SD) file, as described in Section 6.2.1, as input file for the simulation.

Figure 6.1 shows the design of the memory simulation algorithm. The recursive function, needed to simulate the stack behavior (see Requirement 2), is called "allocate()". This function is called when the amount of stack is increasing.

Simulate Stack Allocation In the first part of the algorithm (Part I in Figure 6.1), the time will be delayed as requested by the data set. Then, the stack memory is allocated by calling the alloca()-function and used with the distmemset()-function.

Next Trace Point In the second part (Part II), the algorithm decides whether the stack is increasing or decreasing at the following trace point. It will again reach Part I if the stack is increasing. If the stack is shrinking, the recursion has to be left, which is realized in Part III.

The following stack value cannot be zero as this value would has been handled in Part IV before reaching the Part II.

Simulate Stack Deallocation This is achieved in the third part (Part III). The time delay is simulated and the stack memory is automatically freed when leaving the allocate()-function.

After having left the allocate()-function, the algorithm is either back in the previous function, which initially called the allocate()-function, or it is in Part II

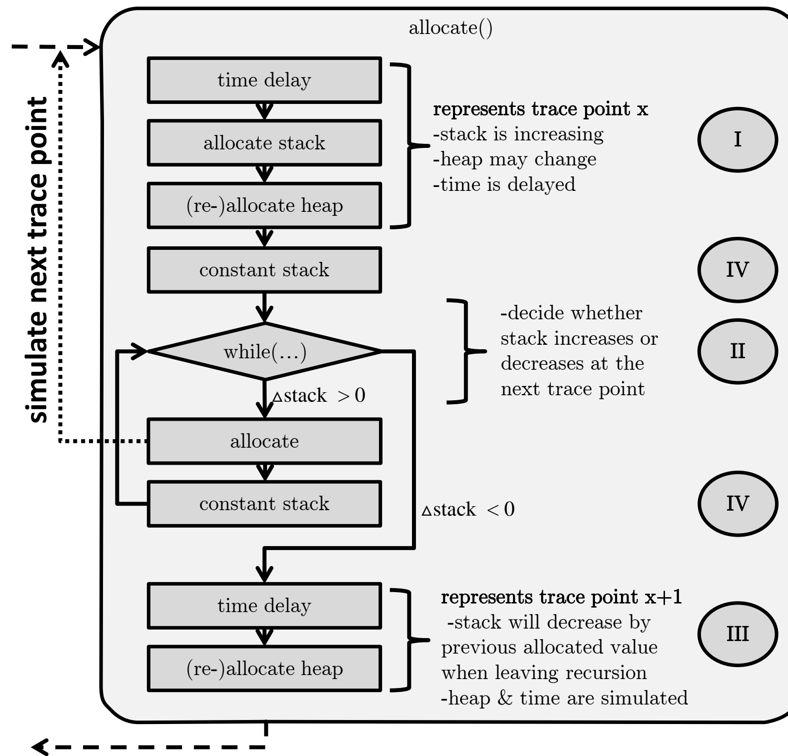


Figure 6.1: Memory Simulation Algorithm

and the while(...) condition proves once more if the amount of stack is increasing at the next trace point. This construction is needed to simulate a sequence of rising and trailing edges.

Simulate Constant Stack In situations where the stack remains constant (see Requirement 4), only the heap and time have to be simulated. This is achieved in both Parts IV.

Simulate Heap (De-)Allocation The heap memory can be de- as well as allocated in the Parts I, III and IV. As it can increase and shrink at these points no specifics have be to considered here (see Requirements 1, 3 & 4).

Use Allocated Memory In order to use the newly allocated memory and, hence, to create the necessary page faults, the distmemset()-function (see Listing 6.3) is additionally called whenever new memory is allocated, i.e., after the “(re-)allocate heap”, “allocate stack” and “constant stack” functions (see Requirements 5 & 6).

The main memory behavior of software modules or functions can be simulated by running this algorithm. All listed requirements from Section 6.1 are satisfied.

The next section presents the *calibration function* for the *mm stubs* algorithm.

6.3 Calibration Functions

This section presents the evaluation of different overhead values, which are caused by executing the *mm stubs* as in Section 6.2.2. After the calculation, the results can be used to improve the main memory simulation algorithm. In the context of *DPS*, they are referred as *calibration functions* (CF).

6.3.1 Measurement Tools

The test environment is based on a Linux operating system (see Section 6.6.1). Hence, measurement tools, which can be used in most Linux systems, are described here. They are used to gather the information to determine the various overhead caused by the main memory simulation algorithm. In this section, the tools used for the *CF* are presented shortly.

Allocated Heap Memory To measure the behavior of the heap memory the "mallinfo" structure provided by the malloc.h header-file is used. This structure delivers several aspects about heap memory allocation such as the memory area, used and free amount of heap memory within the area. The mallinfo struct is read by calling the mallinfo()-function that is also part of the malloc.h header-file.

Allocated Stack Memory The size of the allocated stack memory is measured by reading the base and stack pointer of the running process. Their difference represents the actual amount of stack allocated by the component under test. To read their values with the least overhead, some inline assembler code have been used.

Time The measured values for allocated stack and heap memory have to be put in a chronological sequence. For that reason, the time has also to be measured when picking heap and stack values. Here, the TSC of the system is used.

6.3.2 Overhead Determination

This section presents a possibility to determine the three types of overhead, which are generated by the simulation algorithm: "time", "heap" and "stack".

Time Overhead

When simulating the memory behavior of the component under study the simulation time has to be almost same as the original runtime. But, there are several reasons why the execution time is delayed.

Reasons for Time Overhead There are mainly five reasons for the time overhead produced by the algorithm:

- The execution time of the algorithm without any memory allocation
- The allocation of heap memory
- The use of the allocated heap memory
- The allocation of stack memory
- The use of the allocated stack memory

Running the algorithm consumes time, even if no memory is allocated. This basic time overhead is produced by the algorithm. When allocating heap memory the runtime of the algorithm will increase because of the time that is needed by the realloc()-function. Additionally, the use of the allocated heap memory by the distmemset()-function also takes some time. The same behavior can be regarded when the stack memory is allocated. The alloca()-function and the use of distmemset()-function produces additional time overhead.

Measurement To measure the total time overhead ($time_{overhead}$) for the algorithm several independent measurements have to be done in order to cover all above mentioned cases. For each case, a *CD* file, to simulate the particular situation, is used.

Basic Execution Time First of all, the evaluation of the basic execution time ($time_{basic}$) of the algorithm when no memory is allocated is performed. This is done by running the complete algorithm. To cover the complete algorithm a minimum stack allocation value of 1 byte is used. As the heap should not be regarded within the $time_{basic}$ a value of 0 is applied. The time delay between two trace points is also set to 0 to measure the time consumption of the algorithm.

Measuring the basic time overhead for the algorithm shows a constant offset value for each run. This value is produced by running the algorithm's code. Hence,

it has to be considered as the minimum time resolution that can be simulated by the algorithm. But, because of the other time overhead, e.g., produced by the allocation and use of the memory, this resolution can hardly be achieved.

Time Overhead Caused by Heap Allocation The time overhead produced by the allocation of the heap memory ($time_{heap}$) is caused by the allocation itself and the use of the memory by the distmemset()-function. To evaluate this overhead, both parts of the overhead has to be measured separately. These measurements use an identical *SD* file to simulate heap allocation. The time delay is set to 0 and for stack allocation 1 byte is used at every trace point. This is done to run through the complete algorithm. The amount of allocated heap memory increases at each trace point.

To determine the time used for allocating and using the memory ($time_{heap}^{alloc}$ and $time_{heap}^{use}$) the time before and after the function call is measured as described above. Now, the "after" time is subtracted from the "before" time to evaluate the time delta.

Measuring the time overhead produced by the heap memory allocation highly depends on the architecture and the implementation of the used allocation function as described in [32]. For example, the change of the allocation options within the system influences the number of page faults produced and the way allocated memory is freed within the simulation.

Time Overhead Caused by Stack Allocation To evaluate the time overhead that occurs when allocating stack memory ($time_{stack}$) the same measurements as shown within the heap allocation have to be done. There are two measurements, one to determine the time used to allocate the memory ($time_{stack}^{alloc}$) and another to measure the time to use the allocated memory ($time_{stack}^{use}$). The delay of the stack allocation is determined according to the heap allocation.

The only difference is the *SD* file that is used for the measurement. The time delay is set to 0 also in this *calibration function*. But, there should be no heap memory allocation. Hence, its value is set to 0, too. The amount of allocated stack memory is increasing throughout the *SD* file.

With these two measurements the timing behavior of the stack memory allocation can be determined. This again, strongly depends on hardware and implementation of the system.

Time Overhead in Total In the previous paragraphs, a possibility to determine the various parts of the time overhead has been shown. These times have to be measured separately as different settings for the simulated memory and times have to be used. To get the total amount of time overhead, those parts have to be combined.

$$time_{overhead}(heap, stack) = time_{basic} + time_{heap}(heap) + time_{stack}(stack) \tag{6.1}$$

$$time_{heap}(heap) = time_{heap}^{alloc}(heap) + time_{heap}^{use}(heap) \tag{6.2}$$

$$time_{stack}(stack) = time_{stack}^{alloc}(stack) + time_{stack}^{use}(stack) \tag{6.3}$$

This is presented in Equations 6.1 - 6.3. The “heap” and “stack” parameters denotes the amount of bytes, which will be allocated/used.

Byte Overhead

The execution of the simulation algorithm uses a certain amount of memory. Therefore, the overhead for allocated heap and stack memory have to be measured.

Heap Overhead To measure the heap memory overhead ($heap_{overhead}$), created by the running algorithm, a specific *SD* file with increasing heap memory allocation is used. The time delay is not relevant here and, thus, set to zero. The amount of allocated stack memory is set to 1 byte in order to execute the complete algorithm.

The measurement results show that using the algorithm as described in Section 6 creates only a small overhead in allocated heap memory.

Stack Overhead As done for the heap memory, the overhead of the allocated stack memory has to be determined, too.

The overhead regarding stack memory that occurs when executing the algorithm is mainly caused by two reasons.

On the one hand, there are local variables used to initially set up the simulation. They are located on the stack and, thus, produce a certain amount of allocated stack memory. This is called ($stack_{offset}$).

On the other hand, every function call of the allocate()-function increases the allocated stack memory, which is called ($stack_{overhead}$). This is an important aspect, due to the recursive simulation algorithm, which produces several function calls depending on the *SD* points.

Measuring the stack memory overhead, both $stack_{offset}$ and $stack_{overhead}$, can be done by using a *SD* file that increases the amount of allocated stack continuously. Time and heap values are not considered within this measurement. To estimate the influence of the recursive function calls, the stack memory is allocated several times in a row before freeing it again.

The different measurements show that the amount of additional stack memory allocated by the algorithm ($stack_{overhead}$) is constant for every call of the allocate()-function and, hence, for its recursive call as well.

Overhead Consideration within Simulation Data File Generation

There are several reasons that support and reject the utilization of the overhead during the generation of the *SD* file as can be seen in the following non-exhaustive list:

+ *Calculation Time is Uncritical.*
  When the *SD* file is generated before the execution of the algorithm, the time needed to calculate the overhead that have to be used is absolutely uncritical as it does not influence the runtime of the algorithm.

+ *Execution Time Not Influenced.*
  There is no need of extra execution time during the simulation of the memory behavior of the system, when considering the overhead within *SD* file generation. Using the overhead within the algorithm, would influence the runtime of the simulation as the amounts of time, heap and stack that have to be delayed or allocated, would have to be calculated at first.

+ *No Changes of Algorithm.*
  The simulation algorithm remains unchanged. There are no additional variables needed to calculate the overhead. So, the algorithm needs no additional heap or stack memory as measured within the *CF*.

- *Portability.*
  The *simulated data file* generated only matches to the specific hardware where the *CF* have been performed. For other hardware specifications further *SD* files have to be generated. If the consideration of the measured overhead would be done within the algorithm, the same *SD* file could be used on several platforms.

- *No Error Correction During Runtime.*
  There is no possibility for error correction when executing the algorithm. The data used to simulate the memory behavior is not the original data. It is already adjusted within the *SD* file. Thus, the measurement and evaluation can hardly be done when running the algorithm.

- *Compare Trace File and Original Behavior.*
  Due to the consideration of the overhead within the *SD* file generation, the data of the *SD* file cannot be compared to the original data. Hence, the *SD* file can only be validated by running the algorithm and measuring the results. Regarding the overhead when running the algorithm, would ease the comparison as the *SD* file would represent the original data.

Overhead Calculation when Executing the Algorithm

The advantages and disadvantages of using the overhead within the *SD* file generation are exactly swapped for their consideration within the execution of the algorithm. Hence, they are not closer specified here.

Conclusion

Both approaches have advantages and disadvantages but the possible change in time, stack and heap behavior when using the overhead during the execution of the algorithm is responsible for the use of the overhead calculation within the *SD* file generation. Nevertheless, it should be possible to turn off this overhead calculation within the *SD* file generation and switch over to the algorithm's approach if it is required.

6.4 Simulation Data File Generation

In the previous sections, an algorithm to simulate the heap and stack behavior for a given trace file has been presented. This section presents a possibility to generate a *SD* file considering the *CF*.

6.4.1 Requirements

Due to the recursive algorithm, specific requirements for the *SD* file have to be met. Hence, a closer view on the behavior of the stack memory has to be done and

following cases have to be considered.

1. Simulating Rising Edges With every new call of a recursion and, hence, the allocation of the requested amount of stack, both the base pointer as well as the stack pointer are changed [60]. The base pointer will point to the start of the called function and the stack pointer will point at the end of the allocated memory. When leaving the function and, for this reason, decreasing the depth of the recursion the base pointer as well as the stack pointer will be reset to their previous values. This behavior leads to the fact that, reducing the amount of allocated stack memory by leaving the allocation function will reduce its amount exactly by the value it has been increased when calling this function. Hence, every trace point that causes an increase of the allocated amount of stack has to have its corresponding *SD* point, to free this certain amount of stack.

2. Simulating Trailing Edges When the stack shall be reduced to a certain level, there had to be a *SD* point at this specific amount, previously. Only in this case, the decrease of stack done by leaving the recursive called function can result in the requested amount of stack memory.

3. Simulating Changing Values Another requirement for the *SD* file is that the *SD* points must not contain the total amount of time, stack and heap but their differences to the previous value.

6.4.2 Algorithm

Due to these requirements, an algorithm to generate a *SD* file has been developed. The goal of this algorithm is to find the appropriate *SD* points for each sample point.

An easy approach to calculate a valid *SD* file is to take the stack value from each of those sampled points and to find all interception points in the *measured data* (MD) that have the same amount of stack. By this approach, the requirements of the algorithm are fulfilled, but, there might be a large number of additional *SD* points being created without any need. Having too many *SD* points may result in some performance and timing problems when executing the simulation algorithm because the computation of each *SD* point takes a certain amount of time.

Because of the amount of *SD* points that came up with the first approach, a decrease of the number of *SD* points has to be done. One possibility to achieve this

goal is to change the algorithm, which identifies the data points. There is no need to find all intersection points of the stack values in the *MD* but only one corresponding point that depends on the stack behavior at the position of the measured point. This is presented in Figure 6.2 and will be explained in the following text.

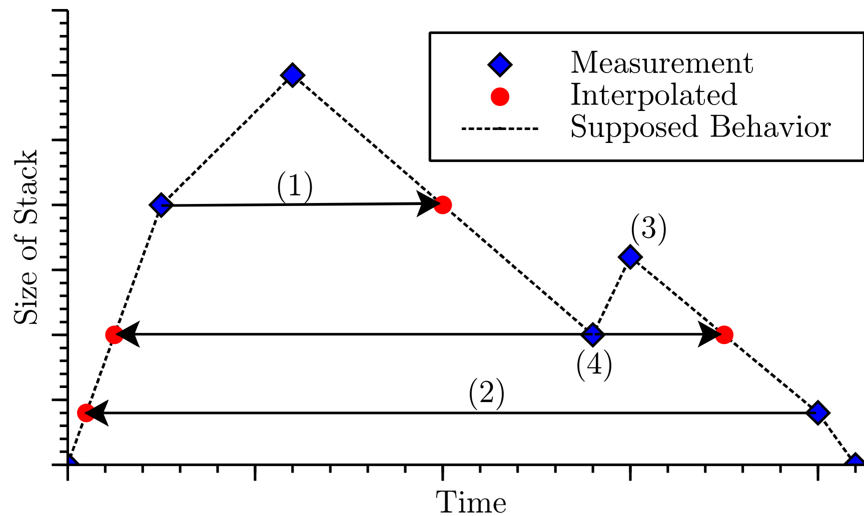


Figure 6.2: Calculating Appropriate Trace Points

Finding the Corresponding Trace Points The *SD* file generator distinguishes the positions of the trace points on the graph. The *MD* points are displayed with "diamond" in Figure 6.2. The interpolated points are represented with an "circle". The dashed line presents the realistic memory allocation behavior of the component under study. This behavior is assumed as the allocation of memory is usually realized in allocating chunks. The time needed to allocate memory is expected to be linear. All calculated data points are called *intermediate simulation data* (ISD).

Rising Edge For trace points being on a rising edge in the graph, the algorithm will search an appropriate intersection on the subsequent trailing edges. This is presented in (1) in Figure 6.2.

Trailing Edge For trace points lying on a trailing edge, e.g., (2), the fitting point has to be found on the previous rising edges.

Maxima For a local maximum, e.g., (3), there is no need to find any corresponding points because of a rising edge followed by a trailing edge will call the recursion and leave it right after the call. So the allocated stack will be freed again

and the total amount of allocated stack will have the same size as prior to the maximum.

Minima For a local minima, e.g. (4), two appropriate *SD* points in the graph have to be found. There has to be one point on the previous rising edges so that the reduction of memory to the local minima is possible; And, another point on one the next following trailing edge. This point is reached after leaving the recursion, which has been started after the corresponding minima.

Conclusion This approach highly decreases the number of trace points created in comparison to the basic approach. Because of this, the amount of overhead produced when executing the simulation algorithm can significantly be reduced.

The determined trace points are often located between two measured points. The value of stack remains unchanged but the time stamp as well as the amount of heap have to be interpolated. So, the calculated points will fit into the supposed behavior as shown in Figure 6.2.

After determining all necessary *ISD* points, the *SD* file exist of trace points with their elements: ISD_{stack}, ISD_{heap} and ISD_{time}.

6.4.3 Simulation Data Point Calculation

This section explains the application of the determined overhead, as described by the *CF*, to generate the *SD* file. Moreover, it evaluates an algorithm to improve the calculation of the *SD* points.

The general equation for calculating the *SD* points is presented in Equation 6.4.

$$SD = ISD - error - overhead() \tag{6.4}$$

The following list describes the separate elements of the equation.

- SD: This is the value, which is used as input for the simulation algorithm. It will be written into the *SD* file.
- ISD: This is the value, which should be simulated according to the calculated *SD*.
- $error$: This is an additional value to consider different errors in the system. These errors are described in the following subsections.

- *overhead*(): This function considers the overhead as measured by the *CF*. The calculation of the overhead is described in the next subsection.

To calculate the *SD* points, an *error* variable has to be subtracted from the *ISD*. This error variable is used for static offset considerations and for the *automatic error correction* (AEC). Additionally, the overhead, determined by the *CF*, have to be subtracted from the *ISD* as these overhead are generated by the algorithm for each simulated trace point value.

Applying the general Equation 6.4 to each simulated value leads to the following equations, which are used to calculate the *SD* variables.

$$SD_{stack} = ISD_{stack} - error_{stack} - overhead_{stack}(ISD_{stack} - error_{stack}) \tag{6.5}$$

$$SD_{heap} = ISD_{heap} - error_{heap} - overhead_{heap}(ISD_{heap} - error_{heap}) \tag{6.6}$$

$$SD_{time} = ISD_{time} - error_{time} - overhead_{time}(SD_{heap}, SD_{stack}) \tag{6.7}$$

Each trace point is written to the *SD* file and consists of: SD_{stack}, SD_{heap} and SD_{time}.

Consideration of the Results of the Calibration Functions

The amounts of allocated stack and heap as well as the time delay have to be adjusted by the measured overhead from the *CF*. Hence, the *CF*' measured data have to be evaluated. In particular, the $stack_{offset}$, $stack_{overhead}(x)$, $heap_{overhead}(x)$ and $time_{overhead}(heap, stack)$ have to be considered.

The equation to calculate the SD value for the stack, i.e., Equation 6.5, is used as an example. From Section 6.3, the following overhead values for the stack have been determined by the *CF*:

$$stack_{overhead}(size) = \begin{cases} 64\text{bytes} & size > 0 \\ 0\text{bytes} & size = 0 \\ -64\text{bytes} & size < 0 \end{cases} \tag{6.8}$$

$$stack_{offset} = 216\text{bytes} \tag{6.9}$$

The *CF*' results (Equations 6.8 and 6.9) are used to generate the *SD*, as shown in Equation 6.10. The $error_{stack}$ variable is initially set to $stack_{offset}$, which is 216 bytes. The measured $stack_{overhead}(size)$ function is used to consider the overhead, which is introduced by the simulation algorithm.

$$\begin{aligned} SD_{stack} &= ISD_{stack} - error_{stack} \\ &- stack_{overhead}(ISD_{stack} - error_{stack}) \end{aligned} \tag{6.10}$$

The results are written into the *SD* file. Running the measurements with this file, which considers the *CF* as well as the errors, show clearly enhanced results.

Automatic Error Correction

An error is introduced in the simulation, e.g., if the *ISD* value is between zero and the overhead value[1]. According to Equations 6.5 - 6.7, the *SD* value would change its sign, e.g., beside of increasing the total value in the system, a decrease would be initiated and, hence, the recursion would be left.

To avoid this error, the *SD* value is set to zero. This means, the trace point is simulated at the minimum level that can be resolved, e.g., the value remains constant for heap or stack values. The error introduced by setting the *SD* value to zero is stored in the *error* variable and will be removed as soon as it is possible, e.g., an *ISD* value greater than the $overhead() + error$ will be allocated for *ISD* values greater than zero.

Figure 6.3 presents the introduced error and shortly depicts its removal. The black graph, marked with "crosses", presents the desired change in the value. The red graph ("squares") shows the behavior, which can be measured in the system by running the simulation.

[1] The *error* value is considered as zero in this example.

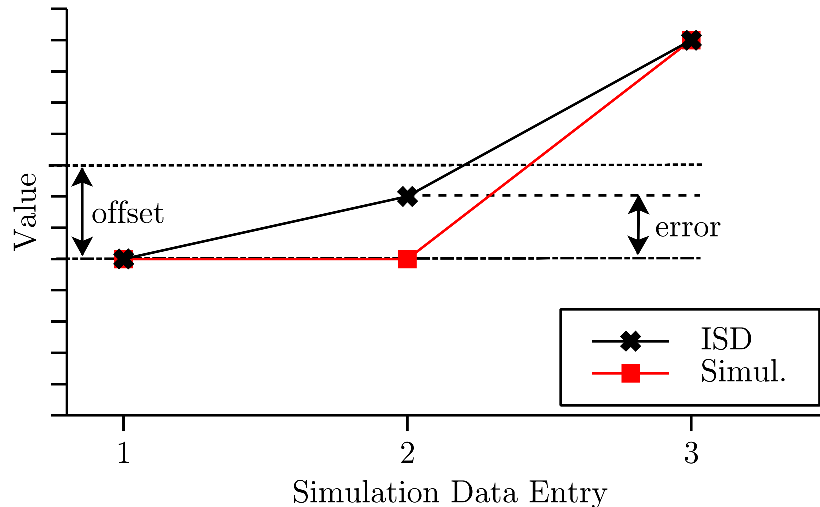


Figure 6.3: Error in Simulation

The error happens at the *SD* entry 2. Here, a value for *ISD* greater than zero and lower than the $overhead() + error$[2] is requested. As described above, in this case the value for the SD is set to zero. Hence, less than the requested value is used by the simulation. The value remains constant and the introduced error is stored in the $error$ variable.

In the next step, which is the *SD* entry 3, a higher value as $overhead() + error$ is requested. Hence, the *ISD* value, the overhead as well as the error introduced at the *SD* entry 2 is considered to calculate the *SD* value. Here, the error is tempered.

Algorithm The following algorithm is used to determine and remove this error:

For each *ISD* a *temporary simulation data* (TSD), starting from the first *ISD*, is calculated. If available, the $error$ variable is initialized to the static offset, as described above. Otherwise, the $error$ variable is set to zero.

$$TSD = ISD - error - overhead() \tag{6.11}$$

The TSD for each *SD* point can be calculated using Equation 6.11. Here, the overhead and the error is subtracted from the ISD value. The TSD value is used to decide whether ISD value can be simulated without introducing an error or not. This is done by comparing the TSD with the ISD value.

Now, the following cases can be distinguished:

[2]The $error$ value at the *SD* entry 1 is zero.

1. $(TSD \geq 0 \wedge ISD > 0) \vee (TSD \leq 0 \wedge ISD < 0)$
In this case, the TSD and ISD values are both greater or smaller than zero at the same time. This means that an error, as described in the AEC, does not occur after this sampling point. The SD value can be calculated using Equation 6.12. Here, the overhead and error is subtracted from the ISD value.

$$SD = ISD - error - overhead() \tag{6.12}$$

Another type of error remains for the heap and stack values. This is caused by calculating the overhead by using $ISD - error$ instead of using SD (see Equations 6.5 & 6.6). But, the overhead depends on the value of the SD point entry, which itself depends on the overhead. As the SD point is not available for calculating the overhead value the result of $ISD - error$, which is close to the SD value, is used. But, an error is introduced as $SD \neq ISD - error$.

$$error = overhead(SD) - overhead(ISD - error) \tag{6.13}$$

This error can be calculated by subtracting the calculated overhead value from the supposed overhead value by using Equation 6.13. Depending on the overhead function, this error can also be zero, e.g., for overhead function as presented in Equation 6.8.

The error does not occur for time values (SD_{time}) as the overhead time only depends on the $heap_{SD}$ and $stack_{SD}$ values, which are available for the calculation (see Equation 6.7). Hence, the $error$ variable is set to zero.

2. $(TSD < 0 \wedge ISD \geq 0) \vee (TSD > 0 \wedge ISD \leq 0)$
This case is taken if a sign flaw, e.g., a stack decrease would happen instead of an increase, would appear. Here, the value of $|overhead() + error|$ is greater than the $|ISD|$. This particularly happens if either an error from an older execution exists and is not removed, yet; And/or a new error appears at this simulation point as explained in this section. Hence, the SD value is set to zero as depicted in Equation 6.14.

$$SD = 0 \tag{6.14}$$

$$error = -ISD + error + min_{sim} \tag{6.15}$$

The *error* is the negative value of the value, which should be simulated, i.e., ISD. The value is negative as our simulated value is smaller than the expected value in case of an increase of the simulated value. Moreover, the remaining error from the last SD point calculation has to be considered. Finally, a variable to consider the minimal possible simulation resolution (min_{sim}) is introduced. This is explained in more detail below. Equation 6.15 can be used to calculate the *error*.

In the algorithm, a variable to consider the minimal possible simulation resolution (min_{sim}) at the SD point entry is introduced. This value is the minimal overhead, which will be used in any case even if the SD point is set to zero. As the algorithm is able to simulate a constant stack and heap size, the $min_{sim} = 0$ for heap and stack. The min_{sim} value is set to $overhead_{time}(SD_{heap}, SD_{stack})$ for time data points (SD_{time}).

The above described algorithm can be used to calculate the SD points. There are two more special cases, which has to be considered for "time" and "stack".

Time The value for the time SD point (SD_{time}) is always greater than or equal to zero as the time cannot be decreased. Hence, the evaluations for $ISD < 0$ in the Cases 1 & 2 of the algorithm does not have to be considered.

Stack The algorithm to simulate the main memory behavior (see Section 6.2.2) has to be recursive. Hence, the stack value cannot be decreased to a certain value but has to be set to the according SD value on the raising edge (see Section 6.4.2). This value is already available in the SD file and can be reused. Additionally, the error, which may have happened at the "increasing" SD point has to be restored, here. Thus, the algorithm as depicted in Section 6.4.3 is not used for decreasing stack SD points.

Conclusion Considering the *CF*, the SD file generation as well as the AEC the SD points can be calculated with an high accuracy. By using the adjustments, which are the overhead improvement and the AEC, the simulation of the stack, heap and time behavior has been improved, significantly. Especially, peaks in allocated memory are simulated very accurate.

6.5 Methodology

During the study of performance behavior of main memory bound systems, normally, a simple indicator that the system under study has a main memory bottleneck is initially given. In general, two basic measurements can be used during the optimization phase:

Page Faults Due to the fact that, an access of secondary memory is very time consuming, this should be avoided. Such an access will be indicated by page faults. Please note that, initially, every first memory access to a page will cause a page fault, to load the data. However, repeated page faults reduce the system's performance by orders of magnitude. In extreme repetition, which is called *page thrashing*, the system will not be operative anymore, because it will be constantly busy by page loading. So the number of page faults can be taken as a measurement whether a system is memory bound or not.

Timing Constraints If either page faults cannot be measured or the real execution time has been chosen as optimization criterion, time has to be taken as measurement, which will be realized by the following definitions:

- t^{CUS}: Time spent in the bottleneck (CUS).
- t^{SUT}: Time spent in the software module or function (SUT), which the CUS is part of.

Based on the optimization measurement, the following methodology can be used:

1. *Determination of the bottleneck.*
   The SUT has to be defined and a suspected bottleneck (CUS) has to be identified, which is done by common SPE methods, e.g., profiling or tracing. According to the chosen measurement method, either the number of page faults or time constraints have to be determined. The measured values have to be statistically viable within several performance test runs.

2. *Create Main Memory Stub.*
   The *mm stub* is now created. First, the functionality of the software module or function has to be simulated using the *simulated software functionality* (SSF). Now, the *mm PSF* will be inserted into the stub. They simulate the memory usage of the CUS. In addition, the time spent in the CUS (t^{CUS}), to guarantee

the original timing behavior, is simulated within the *mm stubs* algorithm. Then, repeated measurements should deliver the same values as the original software.

3. *Validate main memory bottleneck.*
   The next step is a modification of the memory usage parameter in the *mm PSF*. First, the memory usage has to be increased, e.g., 5%, 10%, ..., 100%. If that leads to more successive page faults or a longer system time $t^{SUT}_{stubbed}$, then the CUS is at least a main memory bound bottleneck. Please note that, the values for the initial page faults have to be subtracted as they will still occur in the background at this stage.

4. *Evaluate the optimization potential.*
   Now, different measurements with different optimization parameters have to be realized. Hence, the parameters in the *mm PSF* can be varied. These variations are based on estimations of the performance analyst, e.g., estimations in which parts of algorithm the memory consumption can be optimized. As an example, the memory consumption can be changed by using specialized data structures or by optimizing the data representation itself.

   Additionally, the influence of different memory allocation behaviors can be studied, e.g., by varying the allocation chunk size. For example, the complete data can be statically allocated in the initialization phase or only a subset of the data are allocated in the initialization phase and new data are allocated on demand.

   Moreover, the influence to the time behavior of the bottleneck for optimization strategies based on using data caches can be evaluated. Here, the memory and time consumption can be changed in order to simulate the caching effects.

   The results of this steps are noted for the next step, where a cost estimation of the performance improvement can be carried out.

5. *Cost-benefit analysis.*
   In this step, a cost estimation based on the possible optimization gains should be done. Hence, the amount of effort, which has to be spent for realizing the proposed optimization has to be evaluated. By comparing the possible outcome of the optimization with the necessary effort for this optimization, a cost-benefit analysis can be done.

6. *Optimization of the software.*
   Now, the software module or function has to be optimized. Hence, the results from the cost-benefit analysis can be used for a software improvement related to the optimum between cost and effort.

   Finally, the performance of the software component has to be measured again. A new bottleneck has to be identified (first step), if the results show that the performance targets are not achieved yet.

This section presented a methodology for using *mm stubs* to evaluate the possible outcome of a main memory optimization. This can be achieved by a cost-benefit analysis. The next section evaluates the *mm stubs* by a case study.

6.6 Case Study

This sections discusses the application of the several aspects of the *mm stubs*. First, it evaluates the *CF* for the test environment. Moreover, it validates the *SD* file generation algorithm with its subcomponents: "simulation data point calculation", "calibration functions considerations" and the "automatic error correction" algorithm. To conclude this section, an overall *mm stubs* algorithm validation is provided.

6.6.1 Evaluation Environment

All measurements were performed on a FSC Amilo Si3655 Notebook with an Intel Core(TM)2 Duo P8400 CPU (Intel 64 architecture), which runs with 2.26GHz. As operating system Arch Linux is used. Its kernel version is 2.6.34. The binary has been built using the gnu compiler collection (gcc) without any optimization flags. To guarantee that, the option "-O0" has been used. Beside of running the proof of concept, the system has been idle to avoid further influences on the execution time.

Measurement Tools To offer the possibility to evaluate the simulated behavior of the memory allocation a very precise way to measure the stack and heap allocation has to be used.

For this reason, the value of allocated stack memory is measured by inline assembler calls to read the stack pointer (esp) and base pointer (ebp) registers. The value of ebp is taken at the beginning of the simulation to get a base value for the stack allocation. During the simulation the esp register has been read at every measuring

point. So the offset between the starting ebp and the actual esp gives the actual total amount of allocated stack memory.

To measure the value of allocated heap memory, the mallinfo structure of the malloc.h header-file is read. This structure contains all the desired information about the heap memory for this process.

The *MD* has to be associated with the time spent in the system. Because of this, at every measuring point a time stamp is taken using an inline assembler to read the TSC of the system.

6.6.2 Original Function

According to the needs of the several subsections of this case study, three different *SD* files have been used for validation of the *mm stubs*.

Calibration Functions The *simulation data* file has been generated as described in Section 6.3

Simulation Data File Generation To depict the results of the several algorithms at the best, a small input data file has been used to present the application flow of the generation as well as the error correction algorithms.

Main Memory Stubs Behavior A large *SD* file, which covers the various aspects of the simulation of the main memory behavior has been used to study the *mm stubs*. This file particularly evaluates the simulation algorithm with its different simulation possibilities.

The used *SD* files are described in more detail in the subsections.

6.6.3 Main Memory Stubs

As described above, this subsection is split in to three subparts: *CF*, *SD* file generation and the simulation of the *mm stubs* execution behavior.

Calibration Function

To determine the time, heap and stack allocation offset, created by executing the simulation, the *CF* as presented in Section 6.3 are used. The values for those offset depends on the system's implementation. Hence, the calibration has to be repeated when changing any of the system's parameters. As described in Section 6.3, different *SD* files have to be used to measure the various offset values of the algorithm.

Time Overhead The measurement of the basic time offset has shown to be constant in our setup. It is determined to $time_{basic} = 126775\text{cycles}$[3] with a SCV of 0.009, calculated for 100 test evaluations.

Within this proof of concept, the time consumed by allocating stack and heap memory has been identified. The evaluation of the measurements that were described in Section 6.3, leads to following results for the heap and stack allocation (y describes the previously allocated total memory size in the memory segment and x the newly allocated memory in bytes).

Equation 6.16 is used to calculate the number of page faults at a certain memory allocation value.

$$pagefaults(x, y) = \left\lfloor \frac{(y\%pagesize) + x}{pagesize} \right\rfloor \tag{6.16}$$

The number of bytes, which did not cause a page fault, yet, is calculated by $y\%pagesize$. The result plus the newly allocated memory (x) is devided by the pagesize to determine the amount of page faults for the new allocation. This result is passed to the floor function as page faults can only be a natural number. $Pagesize$ denotes the system page size in bytes.

Time Influence of the Heap Simulation The heap memory will only be reallocated. Hence, the memory value (x) is always greater or equal zero.

As can be seen in Equation 6.17, the time spent for allocating memory heavily depends on whether a page fault is raised in the allocation function or not. Additionally, there is only one page fault in the allocation function even if more than one page is allocated.

$$time^{alloc}_{heap}(x, y) = \begin{cases} 3722\text{cycles} & pagefaults(x, y) > 0 \\ 94\text{cycles} & pagefaults(x, y) = 0 \end{cases} \tag{6.17}$$

$$time^{use}_{heap}(x, y) = 69\text{cycles} * (pagefaults(x, y) + 1) +$$

$$3252\text{cycles} * \begin{cases} pagefaults(x, y) - 1 & pagefaults(x, y) > 1 \\ 0 & pagefaults(x, y) = 0 \end{cases} \tag{6.18}$$

In Equation 6.18, the time spent in the distmemset()-function is calculated. The

[3] I.e., approximately 59μ seconds.

equation consists of two parts. First, the time spent iterating over the memory block, i.e., 69cycles $* (pagefaults(x, y) + 1)$ and, second, the number of page faults occurred in the function minus one as one page fault appeared within the allocation function (see Equation 6.17).

Time Influence of the Stack Simulation For both times, i.e., $time^{alloc}_{stack}(x)$ and $time^{use}_{stack}(x)$, the algorithm does not take significant time to free the stack memory ($x \leq 0$). Additionally, the “freed” memory will not be used, obviously. Hence, both values are set to zero cycles. In the other case, the time needed to allocate and use the new memory can be calculated by using Equations 6.19 and 6.20.

$$time^{alloc}_{stack}(x) = 48\text{cycles} \tag{6.19}$$

$$time^{use}_{stack}(x, y) = 61\text{cycles}+ \\ 3358\text{cycles} * pagefaults(x, y) \tag{6.20}$$

The time to allocate stack memory (Equation 6.19) is constant as only the base- and stack pointer have to be adjusted.

The time spent in the distmemset()-function to initialize the stack memory (Equation 6.20) is the same as in Equation 6.18. The only difference is that the stack allocate function does not raise a page fault. Hence, all page faults are raised in the distmemset()-function. Thus, the total page fault number of the newly allocated memory is used in Equation 6.20.

All the described equations were found by determining the average time stamps of several runs in our test setup and describe the time behavior of allocating and using the heap and stack memory in sufficient accuracy.

When not allocating any heap and/or stack memory at a trace point, the respective times are set to 0. In those cases, they do not have any influence on the calculation of the total time overhead for each measuring point. The equation used to determine the total time overhead $time_{overhead}(heap, stack)$ is described in Section 6.3.

Heap and Stack Overhead As stated in Section 6.3, the heap offset that is introduced when executing the algorithm has to be determined. The measurements showed that $heap_{overhead}$ is constant at 32 bytes, if there is no heap memory allocated within the simulation. If there is any heap memory allocated during the simulation, the heap overhead rises to 40 bytes and also remains constant while the memory is

allocated.

Measuring with the given calibration trace file results in a constant increase of allocated stack per trace point. So, the call of the allocation function allocates a constant amount of stack memory. Because of this measurement, the stack offset is determined to $stack_{offset} = 216$ bytes as well as to

$$stack_{overhead}(x) = \begin{cases} 64\text{bytes} & x > 0 \\ 0\text{bytes} & x = 0 \\ -64\text{bytes} & x < 0 \end{cases} .$$

As these bytes overhead are constant for each execution, there is no need for an statistical interpretation.

The values for stack, heap and time overhead is used to produce a *SD* file. This allows the simulation algorithm to perform a simulation that fit as exact as possible to the desired behavior of memory allocation.

Simulation Data File Generation

The input data file for generating the *SD* file is presented in Figure 6.4 as "measured". Only a small amount of input data is used to depict the application flow of the algorithm at the best.

In Figure 6.4, the "measured" trace points are depicted using a diamond. The x-axis presents the time when the trace point has been taken and the y-axis presents the amount of stack, which were allocated at that time. Here, a scientific trace file has been chosen to cover the different situations as described in Section 6.4.2.

After passing the trace file to the simulated data generation algorithm, the *ISD* are calculated. This is depicted with circles in Figure 6.4 and is called "interpolated". They are calculated based on the assumption about the supposed allocation behavior in the system (blue graph).

As can be seen in the figure, the four different cases, denoted as (1) - (4) in the graph, are accordingly to the assumptions of Section 6.4.2.

The *ISD* points are passed to the *SD* point calculation (see Section 6.4.3). This is presented in Figure 6.5. The axis are according to Figure 6.4.

The blue graph (circles) presents the *ISD* and the red graph (diamonds) presents the *SD*, which will be written into the *SD* file (see Section 6.2.1).

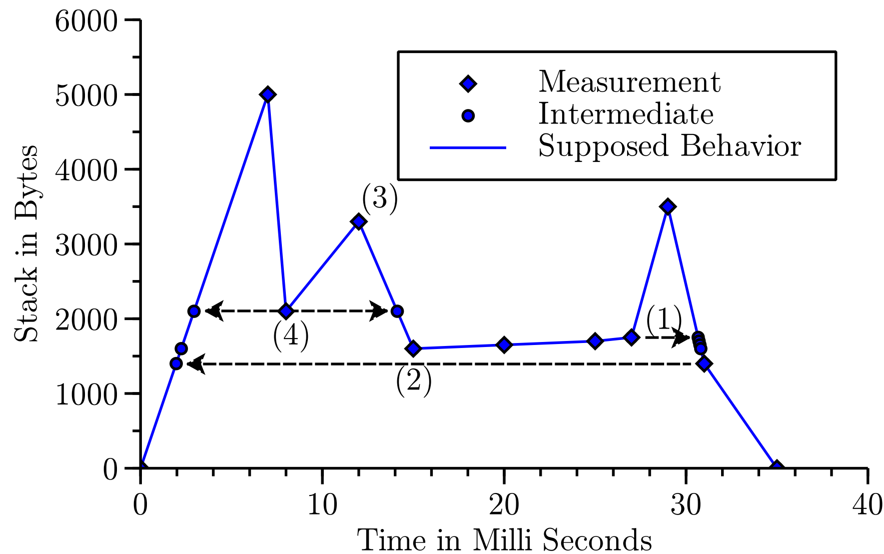


Figure 6.4: Calculation of the *Intermediate Simulation Data*

As can be seen in the figure, the considerations of the *CF* are also taken into account. The consideration of the $stack_{overhead}(size)$ (Equation 6.8) and $stack_{offset}$ (Equation 6.9) is depicted with (1). The results of the $stack_{overhead}(size)$ sums up for each recursion depth. The application of the *AEC* algorithm (see Section 6.4.3) to the *ISD* values is described in (2). Here, a magnification is exemplarily presented for clarification. After calculating the stack value, the time value can be adjusted. Similar as for the stack value, the delta of the time value increases over the time. The time delta is depicted with (3).

As can be seen, the generation of the *SD* file is working properly. Moreover, the *CF* as well as the *AEC*s are included in the algorithm.

Simulation of the *Main Memory Stubs* Behavior

The last step within this case study is the validation of the simulation behavior of the *mm stubs* algorithm. Hence, the *MD* points will be to generate a *SD* file. The result is passed to the simulation algorithm, which is then be executed and measured. Afterwards, the *MD* points are compared to the results of the measurement.

Hence, the *MD* data are described as well as the comparison is done in the following subsections.

Measured Data As input file for *SD* file generation algorithm, the following *MD* data points will be used. To improve the simulation outcome, the results of the *CF*,

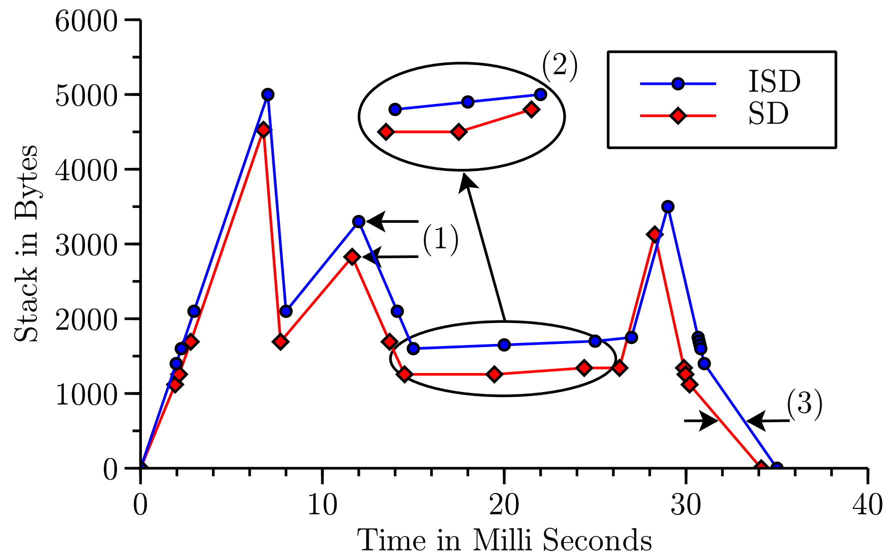


Figure 6.5: Comparison of the *Intermediate Simulation Data* and *Simulation Data*

as described above, are used.

The *MD* are depicted in Figure 6.6. The x-axis denotes the time spent in the software function and on the y-axis the amount of allocated stack memory is presented in kilo bytes. The *MD* has been chosen as all requirements and preconditions can be validated. Hence, the *MD* file has successive de- and allocations (Part 1). Moreover, alternating de- and allocations are included (Part 2). Additionally, data points with constant values on raising and trailing edges (Part 3) as well as slowly increasing stack values (Part 4) are available. Thus, the *MD* points can be used to validate the *CF*, the *SD* file generation algorithm as well as the main memory simulation behavior of *mm stubs*.

The figure presents the *MD* for the stack simulation, only. This is done as the requirements for the heap simulation can be hold easily. Moreover, by removing the heap *MD* from the figure a simplified and, hence, better visualization of the stack *MD* could be presented.

The output is presented as a header file, see Section 6.2.1, containing the data set used within the simulation algorithm.

Measurement After creating a valid header file, an executable of the simulation algorithm can be built. This has been used to execute the simulation of the memory memory allocation behavior.

Figure 6.7 shows the measured stack and heap memory allocation in comparison to the desired behavior. The time in milliseconds is printed at the x-axis and the total

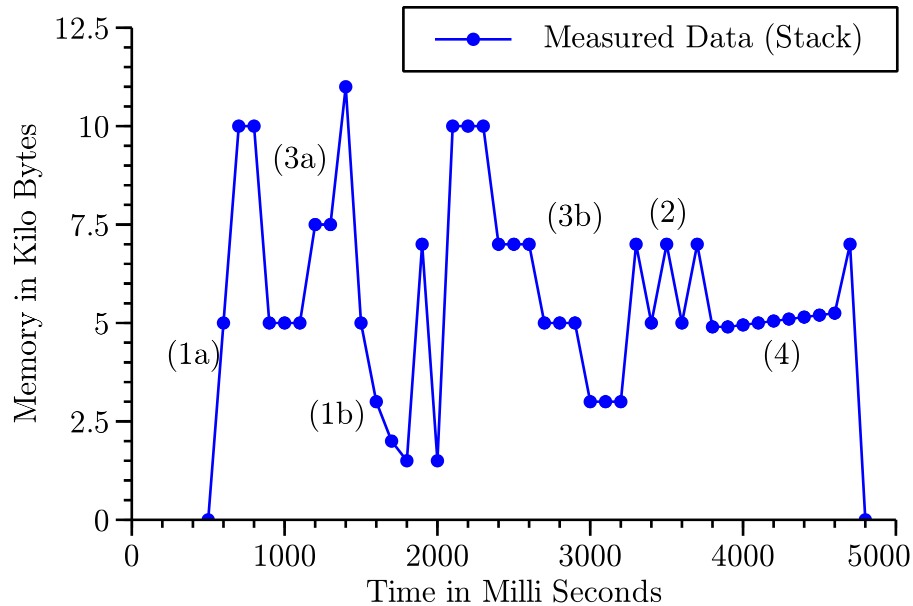


Figure 6.6: Measured Main Memory Behavior

allocated amount of memory in kilobytes is shown at the y-axis. The figure shows the original stack behavior ("Stack Measured Data", blue line), the measured stack allocation during the simulation ("Stack Simulated Data", green dots), the original heap allocation ("Heap Measured Data", black line) and the measured heap behavior ("Heap Simulated Data", red diamonds) while simulating the memory allocation.

Simulating Execution Time When comparing the time supposed by the *SD* file, which is 4.8 seconds, and the execution time measured in the evaluation, which is 4.799300 seconds, it can be seen that the simulation produces only a small variation of time overhead. Here, an overhead of $0.7 * 10^{-3}\%$ in total execution time is produced. So, the total execution time is sufficiently simulated.

Simulating Heap Allocation The analyses of the heap's allocation simulation, as shown in Figure 6.7, depicts that it is very accurate. There is nearly no variation to the desired behavior of heap allocation. It is possible to simulate situations where the heap is rising and falling. Fast switches of allocating and freeing heap memory are simulated exactly. The simulation algorithm works absolutely fine for simulating heap memory allocation in our example.

Simulating Stack Allocation The results of the simulation of stack memory allocation behavior also are satisfying. The allocation behavior can be reproduced

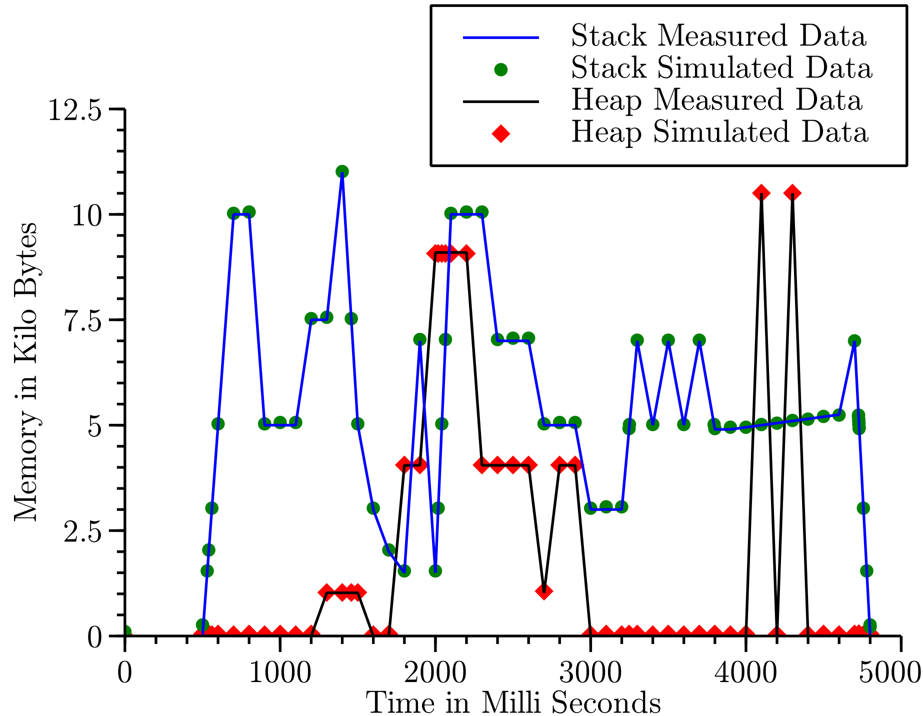


Figure 6.7: Comparison of Original and Simulated Memory Behavior

exactly. Rising and trailing edges as well as constant amounts of stack are simulated in a correct way. High peaks and fast changes of allocated stack are rebuilt as desired. Even slow rises of the allocated amount of stack are simulated quite well.

The measurements were done by using the *mm stubs* algorithm (see Section 6.2.2). The *SD* file has been generated by using the *SD* file generation algorithm (see Section 6.4.2). This algorithm considers the *CF* as well as the two different *AEC* algorithms.

As has been shown, both, heap and stack memory allocation, are simulated with high accuracy and almost without an overhead in execution time. Hence, *mm stubs* can be used to simulate the main memory allocation behavior of software modules or functions.

6.7 Summary

In this chapter, we outlined the *mm stubs*. These stubs are used to simulate the main memory allocation behavior of software functions or modules. Particularly, the heap and stack memory can be simulated by using *mm stubs*.

Within this chapter, some requirements has been defined and the *mm PSF* has been described as well as the *calibration functionality* is explained. Moreover an algorithm to generate *SD* files, which are used as input into the simulation algorithm has been presented. The generation of the *SD* files is supported by two different "automatic error corrections" to improve the simulation results. Finally, a case study, which evaluates the various approaches is included.

The results clearly presents that by using *mm stubs* it is possible to simulate the main memory allocation behavior of software modules and functions. Thus, they can be used in the *DPS* framework to optimize main memory bound systems.

The next chapter describes the *data cache memory stubs* which are one realization of the *memory stubs*.

Chapter 7

Data Cache Memory Stubs

The data cache memory performance simulation functions can be used to model the data cache behavior of software modules or functions. This chapter discusses the requirements, a possible realization as well as a methodology to use these. Moreover, the calibration functions for the data cache memory performance simulation functions are described.

7.1 Requirements

This section specifies the requirements for the *data cache memory stubs* (dcm stubs). They should be able to simulate the data cache access behavior of a software component, in a deterministic way. Moreover, the parameters of the simulation should be adjustable by the performance engineer. Hence, the following requirements have been determined:

1. **Reproducibility.**
   The number of data cache events, which are produced should be deterministic within a specified confidence interval for each test execution. Here, especially the following data cache events should be considered as reproducible:

   - Amount of level one hit accesses
   - Amount of level one miss / level two hit accesses
   - Amount of level two miss / memory hit accesses

   These events should be generated deterministically for read and write data cache accesses.

   A stub, which does not produce an equal amount of cache references might not be usable within the *DPS* framework.

2. **Granularity.**
   The several data cache events should be simulated without significantly influencing further cache events.

3. **Scalability.**
   The amount of data cache events should be adjustable by parameters. In the best case, amount of generated cache accesses should be easily calculated.

4. **Portability.**
   The approach of the *dcm stubs* shall be usable in different hardware architectures.

The requirements cannot be listed by their importance as the importance of the different requirements depends on the developer, who is using the *dcm stub*. E.g., in some cases, the “granularity” requirement might not be important as long as all events can be simulated as needed.

7.2 Realization of the Data Cache Memory Performance Simulation Functions

This section describes a method to produce adjustable amounts of data cache events. The idea of building a *memory stub* for data cache misses is to access data, e.g., a variable inside a program that is not stored in the data cache on the target level and, hence, has to be read from higher levels or main memory. This will cause a cache miss on the target level. A cache miss on a dedicated level will cause either a cache hit on the next level if the data is available on that level or a cache miss if the data is not available. This is only applicable to inclusive caches.

There are different caching architectures, which aim to avoid cache misses as far as possible. In our test system, a n-way set associative cache is implemented. Every location of the main memory can be loaded to n different locations in the cache [17]. The physical address inside the main memory is used to determine the cache set, which stores the data. Hence, it is possible to access data from memory, which will be stored into a dedicated cache set (see [7, 94]).

Each cache set consists of n different cache lines ($assoc$). To permanently create conflict or capacity cache misses the minimum required amount of sequential data ($arraysize$) can be calculated as described by Equation 7.1.

$$arraysize = size + \frac{size}{assoc} \tag{7.1}$$

Any value of $arraysize$ greater than the cache size ($size$) can be used to write into each cache line ($cacheline$), so that the cache is completely filled. This applies to most of the cache replacement policies as discussed below. An $arraysize$ smaller or equal to the cache size will only produce compulsory misses at the first access and cache hits for all subsequent accesses in an otherwise isolated environment. In order to reproducible overwrite a cache set, $assoc+1$ accesses to different data referencing the same cache set have to be done. Therefore, the size of the array has to exceed the size of the cache by the amount of cache size divided by the associativity bytes. This can be achieved by adding $\frac{size}{assoc}$ bytes to the array. Hence, $assoc + 1$ different data references for any cache set are possible.

$Arraysize$ is the optimum size for constantly overwriting cache lines. Any smaller value does not lead to overwrite any cache line periodically. Values greater

than *arraysize* can add additional trashing in other cache levels because the data stub has to load the data sequentially into the higher levels as well. The *arraysize* value provides further advantages such as:

- A smaller amount of memory will be used.
- Runtime improvement of the application, because only data, which has to be referenced, has to be allocated or loaded into higher cache levels. Otherwise, it would lead to a reduction of performance.

However, in some cases an array size greater than *arraysize* might be the best choice because of some side effects, e.g., to create different amounts of cache hits or misses in different cache levels at the same time, to reduce the applications overhead, or to improve the scalability of the amount of created cache events. This strongly depends on the needs of simulating the cache behavior for *dcm stubs* and can easily be realized.

If less or equal than n ($n = assoc$) different data elements, which are stored in the same cache set, are accessed in sequence, no more cache misses other than the first reference will occur. Hence, it is necessary to access at least $n + 1$ different data elements in sequence, which are to be loaded into the same set of cache lines in order to consistently get cache misses during the runtime of the *dcm stub*. The approach is intended to work for read misses as well as for write misses.

This simulation can be used to create a desired amount of cache misses or hits in the target cache level. To achieve this, three different aspects have to be considered:

1. A description of the system's architecture has to be used to adjust the algorithm to the system.
2. The to-be-created events have to be specified. This can be done in a *simulation data* (SD) file.
3. An algorithm has to be determined to create the predefined cache events.

These three elements are described in the following to define the *dcm PSF*.

7.2.1 Architecture Description File

Each level of the caching architecture of the CPU for data, instruction or unified caches can be described by several parameters. First, a general description, like the

cache size (*size*) and the cache line size (*cacheline*) is needed. Second, the cache architecture (see Section 3.4.2), e.g., n-way set associative cache, has to be considered. Thus, the associativity (*assoc*) can be used. Additionally, this specification of the caching architecture has to be provided for each cache level.

```
1  #define LEVELS  2
2  enum cache {L1D=0, L2U};
3
4  struct cacheArch{
5    int cacheline;
6    int assoc;
7    int size;
8    int sets;
9    } cache[LEVELS]={
10   [0].cacheline=64, [0].assoc=8, [0].size=16384,   [0].sets=32,
11   [1].cacheline=64, [1].assoc=8, [1].size=1048576, [1].sets=2048
12 };
```

Listing 7.1: Example Architecture Description File

An implementation of the architecture description file can be found in Listing 7.1. The three parameters, *cacheline*, *assoc* and *size*, are set to the characteristics of each cache level. The different cache levels are defined in Line 1 & 2. First, the number of different cache levels is specified. Second, the different cache levels are provided. Additionally, the set's parameter (*sets*) is introduced for convenience and is used to describe the number of independent cache sets. This parameter can be calculated by $sets = \frac{size}{assoc \times cacheline}$.

The caching architecture of the example can be interpreted as follows. There are two different caches, a first level data ("L1D") and a second level unified ("L2U") cache. Both have a cache line size of 64 bytes and a associativity of 8. Moreover, each of the caches has a cache size and a number of different sets.

7.2.2 Simulation Data File

The configuration of the access behavior of the algorithm is described in the *SD* file. This file specifies particularly the number of events (*accesses*) to the cache level (*cache*) and the access type, i.e., read or write access (*direction*). Additionally, the amount of different cache event types (CET) is specified ($SAMPLES$).

Further parameters, such as *accessstride*, *setstride*, *sets*, *arraysize* and *sleeptime*, are used to configure the access behavior of the algorithm. These are

described in the algorithm section.

```
#define SAMPLES 2
#define ARRAYSIZE_MAX ((1048576*9)/8)

enum direction {READ=0, WRITE};

struct dataset{
  enum cache cache;
  int accesses;
  enum direction direction;
  int arraysize;
  int accessstride;
  int sets;
  int setstride;
  unsigned int sleeptime;
} sample[SAMPLES]={
  [0].cache=L1D, [0].accesses=10000, [0].direction=READ,
  [0].arraysize=18432, [0].accessstride=2048,
  [0].sets=32, [0].setstride=1, [0].sleeptime=0,

  [1].cache=L2U, [1].accesses=10000, [1].direction=WRITE,
  [1].arraysize=ARRAYSIZE_MAX, [1].accessstride=131072,
  [1].sets=2048, [1].setstride=1, [1].sleeptime=1000
};
```

Listing 7.2: Example *Simulation Data* File

In Listing 7.2 an example *SD* file is presented. Here, two different *CET* (see Line 1) are used. The first cache event type is specified in Lines 16 - 18. It is configured to create 10000 the L1 data cache read events (Line 16). The other parameters, which are *arraysize*, *accessstride*, *sets* and *setstride*, are configured to create "miss"[1] events. Here, the whole L1 cache is used. The last parameter of the simulation is *sleeptime*. In this example no delay between the first and second cache event type is used.

More information about the parameters can be found in the algorithm description (see below).

7.2.3 Algorithm

This section describes the algorithm of the *dcm PSF*. The algorithm is split in two parts. The first part is used to process the different *CET* as specified in the *SD* file.

[1] This can only be seen by examining the different parameters.

The second part creates the cache events for this particular cache event type.

```
1  #include"architecture.h"
2  #include"sdfile.h"
3
4  char myar[ARRAYSIZE_MAX];
5
6  void distmemset(char *memory, char init, int allocatesize, long
       int pagesize);
7  void createAccess(int x);
8
9  int main(int argc, char** argv){
10   int x=0;
11   long int pagesize = sysconf(_SC_PAGESIZE);
12   distmemset(myar, 'a', ARRAYSIZE_MAX, pagesize);
13
14   for(x=0; x<SAMPLES; x++){
15     createAccess(x);
16   }
17   return 0;
18 }
```

Listing 7.3: Algorithm to Specifically Access Different Cache Sets (excerpt)

In Listing 7.3 an example of the first part of the algorithm is presented. As described above, the algorithm constantly accesses different memory locations of an array. This array is defined in Line 4. The distmemset()-function as presented in Listing 6.3 is used to initialize the array (Line 12). The *ARRAYSIZE_MAX* value is specified in Listing 7.2 Line 2 and set to the maximum *arraysize* value used within the different *CET*.

In Line 15 of Listing 7.3 the createAccess()-function is called for each cache event type (Lines 14-16). Listing 7.4 presents the implementation of this function. It determines the memory location in the array (*myar*), which is accessed to create the supposed cache event. To create a predefined amount of cache events the "for" loop in Line 14 is used. Here, number of accesses for this cache event type is read from the *SD* file. In Line 16 it is decided whether a read or write event should be created. The according event is then created either in Line 17 for write or in Line 19 for read events. After the event has been created, the algorithm calculates the next memory location (Lines 22-26).

To calculate the next memory location, the parameters *sets*, *setstride* and *accessstride* of the *SD* file (Listing 7.2) are used. These values are described in the following. Afterwards, the calculation of the next memory location is presented.

```
1  void createAccess(int x){
2
3    int i=0;
4    char a='a';
5    int countset=0;
6
7    const int arraysize=sample[x].arraysize;
8    const int sets=sample[x].sets;
9    const int direction=sample[x].direction;
10   const int nextset=cache[sample[x].cache].cacheline * sample[x].
       setstride;
11
12   int access=arraysize-1;
13
14   for(i=sample[x].accesses-1; i>=0; i--){
15
16     if(direction){
17       myar[access]=a;
18     } else {
19       a=myar[access];
20     }
21
22     access-=sample[x].accessstride;
23     if(access < 0){
24       countset++;
25       access= (arraysize-1) - ((countset % sets) * nextset);
26     }
27   }
28   usleep(sample[x].sleeptime);
29 }
```

Listing 7.4: Function to Specifically Access Different Cache Sets (excerpt)

sets This value describes the number of different sets of the cache, which will be used to create the cache events.

setstride The *setstride* value defines the stride between two sets. A *setstride* value of 1 means that the set next to the actual set is used. A value of 2 means that there is one set in between.

accessstride The *accessstride* value is the amount of bytes written into the same set. This is typically set to *size/assoc* but can also be set to whole-number multiples of *size/assoc*. In this case, the distance between two accesses to the same set is higher. This can be used to additionally create dTLB misses.

For each cache event type, the algorithm traverse the array backwards (Line 14-27) to prevent the application from prefetching functions of the CPU [79]. It starts at the last element (Line 12). The distance between two accesses is *accessstride* (Line 22). If the algorithm processed through the whole array, i.e., the next memory location in the array would be smaller than zero (Line 23), the next memory location in the array has to be calculated (Lines 24 & 25). Here, the algorithm starts at the last memory location in the array but subtracts an offset (Line 25).

This offset is used to determine the next set, which will be overwritten and is calculated by the next *sets*' value, which shall be used (*countset%sets*) multiplied by the distance between two sets (*setstride*) and the cache line size (*cacheline*). This calculation is done in Line 10 and called *nextset*.

Finally, in Line 28 the execution between two *CET* can be delayed using the *sleeptime* parameter as specified in the *SD* file (Listing 7.2). Here, the *system non-influencing CPU PSF* is used to simulate the execution time of the bottleneck. The *system non-influencing CPU PSF* was chosen to provide additional CPU time to the system as it should simulate an optimization of the bottleneck. Hence, it can be evaluated whether the system is CPU bound or not as the additional CPU time can be used by other processes. The *sleeptime* is typically set to zero within the execution of one simulation run, i.e., all cache events for the different cache levels and accesses for one test run have been simulated. Between two test runs, the *sleeptime* parameter is adjusted according to the needs of the performance analysis.

In the following an operation breakdown of the algorithm is exemplarily presented in Figure 7.1. This example is based on the following configuration. There is only one data cache with a size of 16 bytes ($size = 16$), cache line size of two bytes ($cacheline = 2$) and an associativity of two ($assoc = 2$). Hence, there are four different sets in the cache. The *SD* entry is configured with the following main parameters: $ARRAYSIZE_MAX = arraysize = 24$, $accessstride = 8$, $sets = 2$, $setstride = 2$ and $accesses = 6$.

On the left hand, the *myar* array is presented. The gray boxes as well as the arrows denote the access sequence of the algorithm. It starts at the last element of the array and advances using the *accessstride* value to traverse the array (Accesses 1-3). Afterwards, it starts at $arraysize - 1 - offset$ and continuous (Accesses 4-6).

On the right hand, the access behavior of the algorithm in the cache is presented. The gray boxes with the numbers depict the cache line, which will be overwritten by accessing the x-element in the simulation. Hence, by the third and sixth access the

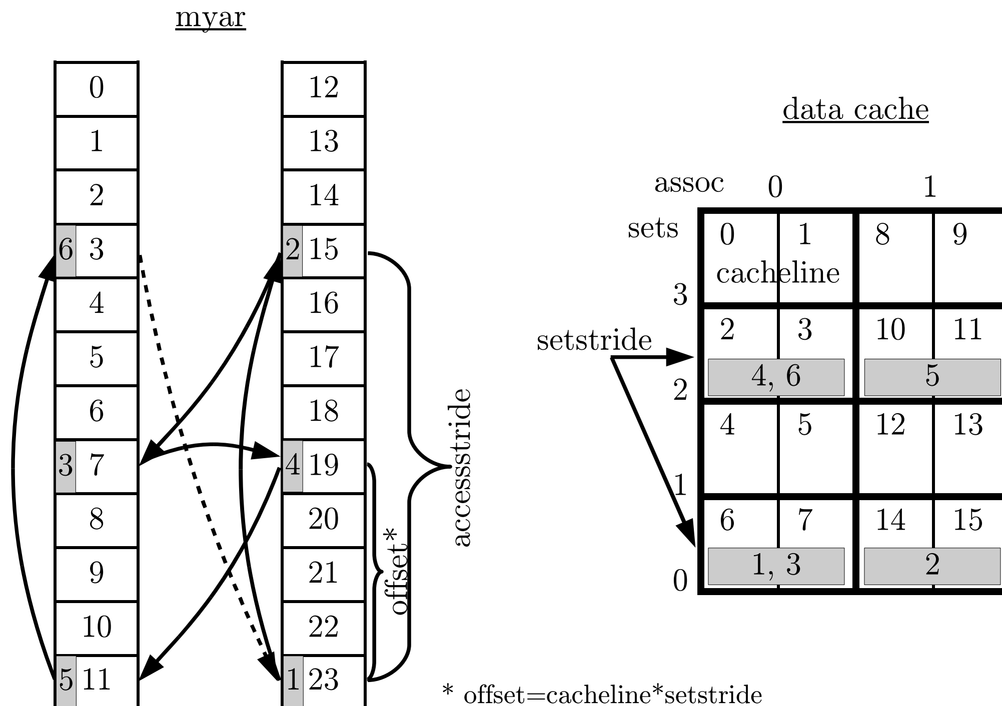


Figure 7.1: Example Access Behavior of the *Data Cache Memory PSF*

cache lines, which have been used by the first and fourth access will be overwritten[2].

Basically, by a misconfiguration conflicting situations can occur. E.g., if the test run is configured to use all *sets* but shall use a *setstride* of two. In this case, the *sets* parameter is more dominant as expected. In the example above, all *sets* are used.

Using the described method for data cache misses on a certain cache level, it produces cache hits in the upper level automatically as long as the amount of constantly referenced data can be contained in the upper level. Otherwise the upper level will also been overwritten. The upper level cache typically has more capacity than the target level, so after the unavoidable first time misses for each accessed data element, the dedicated data cache misses are automatically hits for the higher level data cache.

It is not possible to get hits on higher cache levels without getting misses on lower cache levels because data found in the upper cache level is brought into the lower cache levels. This is the normal behavior for any inclusive cache architecture.

The approach described here can additionally be used to create level one (L1) data cache hits, but the cache architecture has to be modified slightly. Every access

[2]This applies for a LRU replacement strategy (see Section 3.4.2).

to an element of the array has to be inside a single cache line of the L1 cache ($arraysize = cacheline = size$), and the number of sets (*sets*) and the associativity (*assoc*) has to be set to one. Due to the access to different locations within the same cache line, the data is not available in a register but has to be fetched. Hence, it accesses the L1 cache where the data is already available because the cache line has been fetched by the first access to the first byte. An example configuration for a cache is presented as "L1DH" in Listing 7.5.

```
#define LEVELS  1
enum cache {L1DH=0};

struct cacheArch{
  int cacheline;
  int assoc;
  int size;
  int sets;
  } cache[LEVELS]={
  [0].cacheline=64, [0].assoc=1, [0].size=64, [0].sets=1,
};
```

Listing 7.5: Example Architecture Description File to Create Level One Hits

Moreover, the approach can be used for sector caches (see Section 3.4.2) as well. As the "tag" of the memory location for misses is always different to the "tag" stored for the sector, the whole sector is set as invalid and the subsector is filled with new data (see Section 3.4.2). In the case of "hits", the tag will be the same and, thus, no misses are created. Hence, the necessary cache hits and misses can be created using the *dcm PSF*.

To summarize, the *dcm PSF* are able to simulate the data cache memory performance behavior of a software component. This simulation can be done deterministically and can be adjusted by several parameters. Particularly, it can simulate the following cache access behaviors:

- Level one hits
- Level one misses / level two hits
- Level two misses / memory hits

A detailed validation of the approach has been done on the target hardware. The results are discussed in Section 9.4.

7.2.4 Discussion

This section discusses the amount of cache hits or misses occurring during the execution of the *dcm PSF* approach more in detail.

$$ICSR(a_1, a_2) := \exists n \in \mathbb{N}^0 : |a_2 - a_1| = n \times \frac{size}{assoc} \tag{7.2}$$

We define A as a set of main memory addresses, e.g., a_1, a_2, of memory references. Equation 7.2 can be used to determine if two memory accesses referencing the same cache set (*identical cache set references, ICSR*). Two data elements will be stored in the same cache set if their main memory addresses (a_1, a_2) differ by the amount of $\frac{size}{assoc}$ bytes[3]. Here, $size$ denotes the size and $assoc$ the associativity of the cache. Therefore, all multiples $n \in \mathbb{N}^0$ of the distance of two references will be stored to the same cache set.

$$CSR(a) = |\{a_2 \in A | a \neq a_2 \wedge ICSR(a, a_2)\}| \tag{7.3}$$

We define the number of *cache set references* $(CSR(a))$ as the number of independent memory accesses to the same cache set. In Equation 7.3 a formal definition is given. $CSR(a)$ is the number of elements of the set A with different addresses (a, a_2) satisfying the constraints of Equation 7.2.

To describe the possible range of cache events created by the *dcm PSF* approach with arbitrary replacement policies[4] different types of misses according to [7] have to be considered: compulsory ($MISS^{comp}$) and conflict/capacity misses ($MISS^{con}$). The cache will be treated as empty at the start of the application. Therefore, compulsory misses will occur if the application has not used the cache line before. Every first access of data will always create a cache miss. This happens during the first iteration through unique memory locations in the array. If more than a single cache set has been referenced the values given in the equations from this section have to be multiplied by the number of used cache sets. A pass through the memory array is called an iteration (*iterations*). The following cases have to be distinguished:

[3]For more information see also Equation 7.1.

[4]Except of the random replacement strategy as discussed in Section 7.2.5.

- $1 \leq CSR(a) \leq assoc$:

$$MISS^{comp} = CSR(a) \tag{7.4}$$
$$MISS^{con} = 0 \tag{7.5}$$
$$HITS = iterations \times CSR(a) - CSR(a) \tag{7.6}$$

- $CSR(a) > assoc$:

$$MISS^{comp} = assoc \tag{7.7}$$
$$MISS^{con} = iterations \times CSR(a) - assoc \tag{7.8}$$
$$HITS = 0 \tag{7.9}$$

For $CSR(a) \leq assoc$ the independent cache references in the first iteration of the *dcm PSF* loop will produce compulsory cache misses (so Equation 7.4 holds). After this iteration the data which will be referenced are stored inside the cache because there are enough different cache lines (*assoc*) available in the same cache set. All other occurring accesses, therefore, will result in a cache hit on the desired level (so Equations 7.5 and 7.6 hold). This is only partly true for the L1 cache. Here the independent cache set references can also be served from registers, depending on the available registers in the CPU and on the amount of $CSR(a)$. A slightly modified *dcm PSF* approach is additionally described at the end of Section 7 for L1 cache hits. In this case the amount of compulsory misses is 1[5] and the amount of hits is the total amount of references minus the compulsory miss (*cacheline * iterations - 1*).

In the case of $CSR(a) > assoc$, more independent data references will be used than can be stored in a cache set. Therefore, every access to the data element will result in a cache miss (so Equation 7.9 holds). The number of compulsory misses is *assoc* since every cache line in the set will be written (so Equation 7.7 holds). Hence all other references will be conflict/capacity misses (so Equation 7.8 holds).

The amount of cache hits as described in this approach can be less than the adjusted value in real environments. This is expected as other processes running on the CPU will influence the amount of data in the cache. As this is an expected behavior it does not influence the usability of the *dcm PSF* approach for the *memory stubs*. Moreover, the behavior can be used to spot memory problems of the applica-

[5] Assuming that the cache line is aligned.

tion, because the memory accesses are measured and, hence, can be identified. The evaluations done in this sections cover a single cache set. In order to get the total amount of cache references the results from the equations have to be multiplied by the number of referenced cache sets.

7.2.5 Replacement Policies

The deterministic behavior of the approach depends on the replacement policy of the caching algorithm. The following replacement policies will be discussed (see [3, 93]):

- First in first out (FIFO), least recently used (LRU), least frequently used (LFU)
- Pseudo least recently used (PLRU)
- Random Replacement

The amount of cache hits or misses will only be discussed in the following subsections, if they differ from the values described in Section 7.2.4.

FIFO, LRU, LFU

The replacement policy FIFO always overwrites the cache line, which has the longest stay in the cache. Due to the nature of the algorithm, there will always be cache misses when data elements, which are not in the cache are accessed when there are no empty cache lines available. Hence, the algorithm in Listing 7.3 can be used to deterministically create cache misses with every access because it references more cache lines per cache set than can be stored in the cache set. An possible implementation of a FIFO via a round-robin counter, which is incremented after new data is loaded into a set [93], also does not affect the behavior of the approach.

The LRU always overwrites the cache line which has not been used for the longest time. If every cache line is sequentially used every cache line will be accessed by the amount of references. Therefore, there is no difference between the FIFO and the LRU policy in Listing 7.3. The *dcm PSF* approach shows the same behavior for the LRU as for the FIFO strategy and can easily be used for *dcm stubs*.

The same behavior, as described for LRU, can be applied to LFU because of equidistant data access. There will be no cache line, which has more often been referenced by the application.

PLRU

The tree-based Pseudo-LRU (PLRU) as described in [93] approximates the least recently used data. This is realized using tree-bits to point to the (approximated) oldest data stored in the cache set. On sequential accesses to independent data ($CSR(a)$) this approximation gives the same results as the LRU. After *assoc* sequential accesses, the used tree bits point to the cache line, which will be replaced next exactly as the LRU does. Therefore, the *dcm PSF* approach can be used with the PLRU.

Random Replacement

Random replacement strategies, e.g., implemented via a linear feedback shift register (LFSR) [3] will influence the amount of produced cache misses. The bounds of the amount of cache hits and misses, which can occur will be evaluated in this section. Due to the fact that, a random replacement policy is in place, no exact value can be given. Only a range of possible values can be calculated. Relations between the associativity (*assoc*), the total amount of hits (*HITS*) and misses (*MISS*) are given. Furthermore, the misses will be classified into compulsory misses ($MISS^{comp}$) and conflict/capacity misses ($MISS^{con}$) if applicable. Thus, the total amount of misses (*MISS*) is equal to the sum of compulsory misses ($MISS^{comp}$) and conflict/capacity misses ($MISS^{con}$).

For calculating the amount of cache events, this section is split into the first loop ($iterations = 1$) and all successive loops ($iterations > 1$). The equations for loops for $iterations > 1$ will be treated independently from the first loop. Additionally, the evaluations will be done for a single cache set. Hence, the results have to be multiplied by the number of referenced cache sets.

Iterations = 1 The data has not been accessed before, so no cache hits can occur during the first iteration, therefore, all accesses are misses. This is expressed through Equations 7.10 and 7.11.

$$HITS = 0 \tag{7.10}$$

$$MISS = CSR(a) \tag{7.11}$$

As described above, for the calculation of cache misses two kinds of misses can

be distinguished: compulsory and conflict cache misses. To divide the misses and hits into compulsory and conflict/capacity misses and to determine the lower and upper bounds two different cases have to be evaluated regarding the amount of used cache lines ($CSR(a)$) in a cache set:

$1 \leq CSR(a) \leq assoc$: The lower bound for the compulsory misses is one (see Equation 7.12). This means, the referenced data will be stored into any cache line and exactly this cache line will constantly be overwritten by other data references to the same cache set. Therefore, the upper bound for conflict/capacity misses as described in Equation 7.12 is $CSR(a)$ minus one.

$$MISS^{comp} \in [1; CSR(a)] \tag{7.12}$$

$$MISS^{con} \in [0; CSR(a) - 1] \tag{7.13}$$

Because of $CSR(a)$ is less or equal *assoc* each reference can be stored into a different cache line. This defines the upper bound for compulsory misses (see Equation 7.12) and explains the lower capacity/conflict miss bound (see Equation 7.13) as all misses are compulsory.

$CSR(a) > assoc$: In this case the different referenced data do exceed the amount of different cache lines inside of the set. Therefore, the maximum amount of compulsory misses is the amount of different cache lines from one set which is *assoc* and described in Equation 7.14.

$$MISS^{comp} \in [1; assoc] \tag{7.14}$$

$$MISS^{con} \in [CSR(a) - assoc; CSR(a) - 1] \tag{7.15}$$

The maximum amount of cache misses is $CSR(a)$ if every access results in a miss. Thus, the minimum amount for capacity/conflict misses is the total amount of misses ($CSR(a)$) minus the maximum amount of compulsory misses (*assoc*) (see Equation 7.15). The minimum amount for compulsory misses and maximum of conflict/capacity misses has already be described in $1 \leq CSR(a) \leq assoc$ and is one and $CSR(a)$ minus one (see Equations 7.14 and 7.15).

Iterations > 1 This section evaluates the amount of cache misses for all *iterations > 1* of the inner loop as described in Listing 7.3. The iterations will be evaluated without the effects of the first initial loop iteration, which has been described above. For convenience, we abbreviate $i = iterations$. All occurring misses

will be conflict/capacity misses, for simplicity they will be called misses. As already described in Paragraph "*Iterations = 1*" this section will also be split into two cases:

$1 \leq CSR(a) \leq assoc$: Due to the random nature of this replacement policy and the fact that there are more cache lines in one set available as data will be referenced, all occurring cache events can be either hits[6] or misses[7] (see Equations 7.16 and 7.17) in each loop. The maximum amount of possible hits and misses is the number of hits and misses per loop multiplied by the number of loop minus one, because the first loop is evaluated separately. So Equations 7.16 and 7.17 hold.

$$MISS \in [0; (i-1) * CSR(a)] \tag{7.16}$$

$$HITS \in [0; (i-1) * CSR(a)] \tag{7.17}$$

This is a worst case for the *dcm PSF* approach because, there is almost no possibility to determine the amount of hits or misses in advance. Hence, the approach cannot be used in this scenario.

$CSR(a) > assoc$: The maximum number of cache misses, which can occur for a single iteration other than the first one is the amount of initial independent cache references per cache set ($CSR(a)$) as explained for the first loop. This results have to be multiplied by the number of iterations minus one as already explained. The minimum amount of cache misses can be calculated by subtracting the maximum amount of possible hits from the total amount of references ($(iterations - 1) * CSR(a)$). This leads to Equation 7.2.5.

$$\begin{aligned} MISS &\in [(i-1) * (CSR(a) - assoc); (i-1) * CSR(a)] \\ HITS &\in [0; (i-1) * assoc] \end{aligned} \tag{7.18}$$

The maximum amount of cache hits will be achieved if every access will be a hit; therefore, the maximum amount per loop is *assoc*. The minimum number of hits is achieved when the maximum number of misses are generated during a loop iteration, so if every access is a miss. When the maximum number of misses is equal to the number of accesses, the minimum number of hits is equal zero. The lower and upper bound for all iterations beside of the first is presented in Equation 7.18.

As can be seen from the equations in the best case the number of misses is equal to the number of accesses. This is equal to the behavior when another deterministic

[6] All data references to different cache lines.

[7] All data references to the same cache line.

replacement strategy, e.g., LRU is used.

To get the total number of possible hits or misses the minimum as well as the maximum values of cache hits and misses, of the Sections *iterations* $= 1$ and *iterations* > 1 has to be added and finally multiplied by the number of referenced cache sets. This provides the total range of misses and hits which can occur using the *dcm PSF* approach in caching architectures which uses the random replacement policy.

Conclusion

Whereas, the approach is working for most of the caching architectures, the replacement policy strongly influences the usability of the approach. Due to the unpredictability of the random strategy, the results, as evaluated in Section 7.2.5, depicts a worst case scenario, where it is only possible to determine the lower and upper bounds of the numbers of cache misses or hits.

In our working environment an Intel Pentium 4 processing unit will be used. Therefore, the replacement strategy is a PLRU. This means that our approach is valid in this environment.

7.3 Calibration Functions

The *CF* are used to adjust the various simulation possibilities to the system. Hence, the system behavior while executing a predefined simulation configuration is studied. Here, the results of the measurement will be compared to the supposed values. This can be used to adjust the parameters of the *dcm stub* to achieve the desired data cache access behavior.

The following steps have to be done in order to calibrate the *dcm stubs* to the system:

1. *Determine hardware.*
   The capabilities of the hardware architecture have to be determined regarding the needs of the *data cache memory stubs*. This consists mainly of two points: "caching architecture" and "measurement capabilities".

   (a) The caching architecture of the system has to be determined, e.g., by using hardware specification documents. Particularly, the different cache types and cache levels have to be determined as well as the cache replacement policy. The machine description language, as specified in [82], can be used for tracking the results of this study.

(b) The second part of this step is to determine the measurement capabilities, e.g., by using the hardware specification documents. Usually, the amount of cache events caused by the application can be determined using hardware counters (see Section 2.5.1). If the measurement capabilities of the hardware are not sufficient, further evaluation techniques should be considered, e.g., using simulation tools (see Section 2.5.3) or by a binary analysis.

This step provides information whether the specified *dcm PSF* can be used as the usability strongly depends on the replacement policy. Moreover, it provides a sound overview of the used architecture as well as the measurement possibilities.

2. *Determine use case.*
The several cache events, which shall be simulated by using the *dcm stub*, have to be determined, e.g., the stub shall simulate different amounts write hits in each cache level.

The caching architecture, which has been determined in Step 1a, can be used for a breakdown of the several *CET*. From the example above, this would lead to L1 write hits and level two (L2) write hits[8].

The separated cache events will be examined independently by using "dedicated *dcm stubs*". Hence, the next step has to be redone for each identified cache event.

3. *Study simulation behavior.*
In this step, the influence of each "dedicated *dcm stub*" to the system has to be measured. It consists mainly of two different evaluations:

First, the influence of the stub to the desired cache event has to be evaluated. And, second, the amount of cache events, which shall not be produced by the stub has to be studied. E.g., to create L1 hits the required data has to be transfered into the L1 of the cache. Hence, they have to be available in the upper cache levels first for inclusive caches (see Section 3.4.2).

The measurements within this step have to be repeated for different input parameters, e.g., the supposed amount of the cache events as well as different cache line access patterns should be evaluated.

[8]These cache events are the same as L1 write misses.

The result of this step are different graphs showing the amount of measured cache events depending on the preset values of the *dcm stubs*. Hence, the parameters for the simulation of the cache event as well as the further influences to other cache levels can be determined.

4. *Calculate functions.*
   Here, calculation functions are determined to calculate the input parameters for a *dcm stub*.

   Hence, each all determined function for each “dedicated *dcm stub*” from the step above have to be separated according the different cache events. Now, the equations for each cache event have to be combined to calculate the input parameters for the *dcm stubs*. As higher levels of the cache also influences the lower levels, the calculation of the input parameters shall be done for higher cache levels first.

   The different “dedicated *dcm stub*” are combined to a single *dcm stub* using the calculated parameters.

The provided steps within this section can be used to evaluate the usability of the *dcm PSF* on the architecture. Based on the evaluation results, the *dcm stubs* can furthermore be adjusted to the system in order to gain valuable simulation results. This is commonly referred as *CF* within the *DPS* framework. The next sections evaluates a methodology for using *dcm stubs* to optimize the performance of the software module or function.

7.4 Methodology

This section describes a methodology to use the *dcm stubs* within the *DPS* framework. It applies the *DPS* methodology of Section 4.6 to the *dcm stubs*. This methodology can be used to optimize software modules of cache memory bound systems.

1. *Determine CUS:*
   As a first step of any optimization, the timing behavior of the system in total has to be examined. Then, one component of the system (CUS) has to be chosen, which seems to be a bottleneck [55, 121]. Several performance measurements have to be done until the results seem to be deterministic within a given confidence interval. The following values has to be deterministic:

t^{CUS} the time, which will be used in the CUS.

t^{SUT} a performance indicator of the module or function. This value will be measured to validate the systems performance in total. Often it is the time used for the completion of a service.

$L1^{CUS}_{readhit}$ a counter where the number of L1 cache read hits caused by the CUS has been measured.

$L1^{CUS}_{writehit}$ a counter where the number of L1 cache write hits caused by the CUS has been measured.

$L1^{CUS}_{readmiss}$ a counter where the number of L1 cache read misses caused by the CUS has been measured.

$L1^{CUS}_{writemiss}$ a counter where the number of L1 cache write misses caused by the CUS has been measured.

$L2^{CUS}_{readmiss}$ a counter where the number of L2 cache read misses caused by the CUS has been measured.

$L2^{CUS}_{writemiss}$ a counter where the number of L2 cache write misses caused by the CUS has been measured.

Please note that, the way to determine the values of L1 and L2 parameters strongly depends on the available tool set. As an example, the OProfile tool set can be chosen. Additionally, further cache events, e.g., level three read hits, can be used if applicable. This step has to be repeated a couple of times in order to get some statistical distribution of the required values. If there is only a small variation, we can go further.

2. *Create DCM Stub:*
   After creation of the functional stub, which covers the functional behavior, a performance stubs has to be created, according to Section 7.2.

   So, the above listed values will be simulated by the component. Please note that some values can only be approximated because of the construction of the cache, e.g., the cache line size. As last step, the runtime behavior of the CUS has to be simulated using the *PSF* of the *CPU stubs* (see Section 5).

3. *Validate DCM Stub:*
   The measurement results have to be validated. The value of t^{SUT} for the CUS has to be equal to the t^{SUT} of the *dcm stub*. Whenever, there is a significant

difference, further analysis of the memory stub has to be done; e.g., it might be possible that the CUS is too big and has to be decomposed.

4. *Evaluate optimization potential:*
   Now, a first estimation of the CUSs optimization has to be validated. Normally, the developer of this component has rough estimations of the performance improvements which will be used for further studies. If not, he will simply reduce the value of cache usage by e.g., 10% and start a measurement. Therefore, he simply reduces the number of L2 accesses and, furthermore, the number of L1 accesses. The determined time will be called $t^{SUT}_{improved}$. This measurement is an approximation for the improved system behavior after optimization. It should be repeated with further reductions of the number of access operations.

   Based on the measurement results, an ideal target value for memory optimization can be determined and a cost estimation of the improvement function can be done. Now, a detailed cost-benefit analysis is possible to determine the ideal optimization factor. This will lead to a solid basis for a performance optimization.

5. *Optimize software bottleneck:*
   Based on the measurement results of Step 4 the optimization of the SW module can be started. The proper working can be validated by measuring t^{SUT} afterwards.

   If the values are equal, then the optimization has had the desired effect and the next optimization can be done (Step 6). If not than either the optimization environment or the *dcm stub* does not behave correctly and has to be modified (Step 2).

6. *Verify systems' performance:*
   After the optimization, determine whether the application will reach the desired performance targets. If the applications has to be optimized further, determine a new CUS and go to Step 1.

In some HW environments, the determination of some values of Step 1 is not possible. In this case a different approach can be chosen by simply adding additional memory needs. Instead of replacing the CUS by a *dcm stub*, the original code will be kept. Only the *dcm PSF* will be included. With this setup, measurements can

be achieved, where any additional need of memory can be checked. Whenever t^{SUT} afterwards increases, then the system is at least memory bound. So, increasing the memory or optimization of memory needs will be required. However, with that approach no cost-benefit analysis or a hidden bottleneck detection can be achieved, which is the case if the before proposed method can be chosen.

7.5 Case Study

This section evaluates the proposed *PSF* for the *dcm stubs*. This is done by a case study. Hence, the cache access behavior for the different cache event types is studied. The next subsections describe the test environment and the original functionality. This is followed by the evaluation of the *dcm stub*.

7.5.1 Evaluation Environment

The equipment of a telecommunication network has been used to validate the measurements. It hosts a 2.8 GHz Intel Pentium 4 central processing unit with hyper-threading disabled. The operating system is a standard Linux running on a 2.6.22 kernel. The system has separated data and trace (similar to an instruction cache) caches on L1 and an unified cache at L2. The caching architecture of the processor is described according to [82], the values have been derived from [15, 51] and verified on the target:

```
1  MemoryArchitecture =
2  L1D [     256, 64, 8, *, L2U,  2],
3  L2U [ 16384, 64, 8, *, Mem, 18];
```

Listing 7.6: Caching Architecture

In Listing 7.6 the architecture of the target is described. Each lines starts with the name and type of the available caches, e.g., L1D means a L1 data cache and L2U describes the unified L2 cache. Each level of the cache hierarchy is described using the following items:

1. The amount of different available cache lines, i.e., $sets \times assoc$.
2. Cache line size ($cacheline$).
3. Associativity of the cache ($assoc$).

4. Bandwidth between the current and the lower level on a miss (bytes/cycles). This value has not been used (*).

5. The next level in the memory hierarchy.

6. The cycles needed to access this caching level. The time is described for integer operations.

The values described in this list match the parameters as defined in Listing 7.1. All evaluations of this section have been done using either the configuration for the L1 cache (see line “L1D” in Listing 7.6) or L2 cache (see line “L2U” in Listing 7.6). Only the additional required parameters, as described in Section 7.2, will be described in the following section.

To determine the necessary values as described in Section 7.4 the hardware counters of the CPU have been used and accessed through OProfile[9]. Some values could not be measured by OProfile. Callgrind[10] measurements as well as binary analysis were applied in this case.

We ran the tests with executables generated by the GCC compiler (GCC version 4.2.1). Optimization flags were not used to compile the stub to avoid optimizations from the compiler.

7.5.2 Original Function

The application, which has been used for the case study, is based on matrix multiplication as described in [100]. Three two dimensional arrays have been used to store two input matrices and one output matrix. The size of a row as well as for a column was configured to a value of 1024 bytes. So, each matrix has a size of one mega byte, which is close to the size of the second level cache on the used testing system. The calculation inside of the algorithm has not been optimized, i.e., no array transformations has been done. The determined values for different characteristic parameters of the original CUS are given in Table 7.1.

As the used application is completely stubbed, the times t^{SUT} and t^{CUS} are equal. Therefore, the t^{SUT} time will not be mentioned separately. The Intel Pentium 4 CPU does not provide a write miss counter for the L1 cache. Hence, these events cannot be measured by OProfile. An binary analysis of the CUS has shown that the used

[9] OProfile - A System Profiler; http://oprofile.sourceforge.net/news/
[10] Callgrind is part of Valgrind. See:
http://valgrind.org/info/tools.html

t^{CUS}	28.28s
$L1^{CUS}_{readhit}$	1,147,703,472
$L1^{CUS}_{writehit}$	1,073,741,824
$L1^{CUS}_{readmiss}$	2,073,486,000[11]
$L1^{CUS}_{writemiss}$	0
$L2^{CUS}_{readmiss}$	36,000[12]
$L2^{CUS}_{writemiss}$	0

Table 7.1: Measured Memory Performance Behavior of the Original Function

algorithm performs a read access to all data followed by a write access to these data. As read accesses bring data to the first level cache all write accesses result in first level cache hits. Neither $L1^{CUS}_{writemiss}$ nor $L2^{CUS}_{writemiss}$ occur in the system. Callgrind has been used to determine the L1 read and write hits as the architecture does not provide appropriate hardware counters. The remaining values, $L1^{CUS}_{readmiss}$ and $L2^{CUS}_{readmiss}$, of Table 7.1 have been measured with OProfile. As it is a sampling based profiling application, the measurements have been done ten times in order to evaluate the statistical behavior of the occurred events using the SCV.

7.5.3 Data Cache Memory Stubs

According to the values of Table 7.1, the parameters for the stub have been adjusted. As the matrix multiplication uses the three two-dimensional arrays with a size of one mega byte the complete cache for L1 and L2 will be used. Therefore, the stub will also be set to use the whole cache by setting its *sets*, *setstride* and *accessstride* parameters to according values. The *arraysize* and $ARRAYSIZE_MAX$ values were chosen according to the evaluations presented in Section 9.4. These results have also been used as *calibration functions*. The different parameters for the cache event type simulation are described in Listing 7.7.

The L1 and L2 write events are not simulated as these events do not occur in the original function (see Table 7.1). Moreover, the *sleeptime* parameters are set to zero. This is because the execution time of the *dcm stub* is equal to the execution time of the CUS. More information about the time behavior is presented below.

The caches were modeled as described in the architecture files (see Listing 7.1 & 7.5) to create the necessary cache events. The architecture file (see Listing 7.7) lists the following caches: “L1D”, “L2U” and “L1DH”.

[11]The SCV is 0.00000353.
[12]The SCV is 0.00617284.

```
1  // Architecture File
2  #define LEVELS  3
3  enum cache {L1DH=0, L1D, L2U};
4  struct cacheArch{
5    int cacheline;    int assoc;   int size;          int sets;
6  } cache[LEVELS]={
7    [0].cacheline=64, [0].assoc=1, [0].size=64,       [0].sets=1,
8    [1].cacheline=64, [1].assoc=8, [1].size=16384,    [1].sets=32,
9    [2].cacheline=64, [2].assoc=8, [2].size=1048576, [2].sets=2048
10 };
11
12
13 // Simulation Data File
14 #define SAMPLES 4
15 #define ARRAYSIZE_MAX (67108864)
16 enum direction {READ=0, WRITE};
17 struct dataset{
18   int cache;          int accesses;
19   int direction;      int arraySize;
20   int accessStride; int sets;
21   int setStride;      unsigned int sleeptime;
22 } sample[SAMPLES]={
23   // L2URM
24   [0].cache=L2U,               [0].accesses=36000,
25   [0].direction=READ,          [0].arraySize=ARRAYSIZE_MAX,
26   [0].accessStride=131072, [0].sets=2048,
27   [0].setStride=1,             [0].sleeptime=0,
28   // L1RM/L2RH
29   [1].cache=L1D,               [1].accesses=2073486000,
30   [1].direction=READ,          [1].arraySize=32768,
31   [1].accessStride=2048,       [1].sets=32,
32   [1].setStride=1,             [1].sleeptime=0,
33   // L1WH
34   [2].cache=L1DH,              [2].accesses=1073741824,
35   [2].direction=WRITE,         [2].arraySize=64,
36   [2].accessStride=1,          [2].sets=1,
37   [2].setStride=1,             [2].sleeptime=0,
38   // L1RH
39   [3].cache=L1DH,              [3].accesses=1147703472,
40   [3].direction=READ,          [3].arraySize=64,
41   [3].accessStride=1,          [3].sets=1,
42   [3].setStride=1,             [3].sleeptime=0
43 };
```

Listing 7.7: *Simulation Data* File (Case Study)

Validation The stub's performance data has been measured in the same way as within the original function. The results are provided in Table 7.2 and discussed below.

t^{STUB}	28.75s
$L1^{STUB}_{readhit}$	1,147,703,472
$L1^{STUB}_{writehit}$	1,073,741,824
$L1^{STUB}_{readmiss}$	2,060,992,200[13]
$L1^{STUB}_{writemiss}$	0
$L2^{STUB}_{readmiss}$	36,000[14]
$L2^{STUB}_{writemiss}$	0

Table 7.2: Measured Memory Performance Behavior of the *Data Cache Memory Stub*

The comparison of the values has been done in three stages. First, the oprofile measurements are discussed for the L1 read misses / L2 read hits and L2 read and write misses. Second, the results of the callgrind evaluation are presented. Finally, the time behavior of the original function and the *dcm stub* is evaluated.

Oprofile Measurements Three different cache event types were measured using Oprofile.

First, the evaluation is done for the L1 read misses / L2 read hits, which are denoted as $L1^{STUB}_{readmiss}$. The amount of cache references, which are created by the original function for this cache event type is 2,073,486,000. The measurements of the stub provided an average result[15] of 2,060,992,200 cache events. This is approximately 0.60% less than the events created in the original function.

Second, the L2 read miss events are studied. Here, the measurements of the original function provided a result of 36,000 events. The average value measured in the stub is 6 samples, which is also about 36,000 events.

Third, the L2 write miss events are evaluated. In both cases, the original function and the execution of the *dcm stub*, 0 samples have been measured.

All results within this evaluation pointed out that the according cache event type can be simulated with high accuracy.

[13]The SCV is 0.0000010.
[14]The SCV is 0.012.
[15]As described above, 10 test runs has been done.

Callgrind Evaluation The L1 read and write hits were evaluated using callgrind as described above. As callgrind simulates a CPU including the caching architecture several insufficiencies are introduced in the measurements, e.g., no write buffer is simulated. Because of these insufficiencies, the *dcm PSF* creates exactly the specified amount of events in callgrind. Hence, it is not surprising that the events measured in the original function does not differ from the results of the stub. Nevertheless, callgrind can be used to evaluate the cache memory behavior of applications. Thus, the *dcm PSF* simulated the cache type events correctly.

Time Evaluation The time spent in the original function is 28.28 seconds on average. The time spent in the stub is 28.75 seconds. Hence, the stub takes slightly more time than the original function (+1.66%). This is not surprising as the original function is almost complete memory bound. The stub instead has some CPU "intensive" instructions. Moreover, the time spent in the memory handling remains the same in both applications.

Conclusion As can be seen from the comparison of the measured values for the original function (Table 7.1) and the *dcm stub* (Table 7.2), the characteristic parameters are rebuild almost exact. Hence, the *dcm stub* is able to simulate the cache access behavior of a software function or module with high accuracy.

7.6 Summary

This chapter outlined the *data cache memory stubs.* These stubs are used to simulate the cache access behavior of software modules or functions. Especially, L1 and L2 hit and miss events for read and write accesses can be simulated.

Within this chapter, some requirements have been defined and the *data cache memory performance simulation functions* have been described as well as the *calibration functions* are explained. Finally, a case study, which evaluates the various approaches is included.

As can be seen by the results, *data cache memory stubs* can be used to simulate the cache access behavior of software functions. Thus, they can be used to optimize data cache memory bound systems by using *dynamic performance stubs.*

The next chapter describes the *simulated software functionality.* It can be used to simulate the software functionality of algorithm. Hence, the functional behavior of the bottleneck can be rebuilt by using the *simulated software functionality.*

Chapter 8

Simulated Software Functionality

The simulated software functionality allows the replacement of an existing software module or function by a stub, in order to do software performance improvement studies. This chapter covers the major aspects of the simulated software functionality. First, requirements are defined. Moreover, a methodology for using the simulated software functionality is presented. Afterwards, a possible realization is described. Finally, a case study, which applies the methodology to the realization of the simulated software functionality concludes the section.

8.1 Requirements

In order to be able to replace a bottleneck with a *dynamic performance stub*, it is necessary to recreate the functionality of the software module or function. Hence, the following requirements can be defined and subdivided into: *requirements on the system*, which has to be satisfied by the system under test and *requirements on the simulated software functionality*, i.e., how the *simulated software functionality* (SSF) has to behave:

1. *Basic Requirements on the system:*

   (a) **Deterministic CUS behavior.**
   The software module or function has to have a deterministic functional behavior. I.e., it has to produce the same results within each equivalent execution, e.g., deterministic output values depending on the input values. Another possibility is that the function can also return random values, if this is the specified behavior within the system. The execution time of the function has to be deterministic, too.

   (b) **Reproducible test execution.**
   The used test environment and scenarios have to deliver reproducible results. This is a common requirement to any test environment.

   (c) **Automated test case execution.**
   It is preferable if the test cases can be executed automatically. This property significantly reduces the effort for doing test executions repeatedly. Additionally, reproducible test scenarios can also be used for performance measurements.

2. *Requirements on the SSF:*

   (a) **Automatic generation of the serialization specification.**
   The serialization specification shall be generated automatically. This removes additional effort for the user of the *SSF* and to decrease the amount of possible errors, e.g., writing a wrong serialization specification. Hence, a serialization functionality shall be provided as well as an almost automatically serialization specification shall be generated. These can be used to automatically store the C++ objects. A fully automatic generation of the serialization specification is not feasible as explained in Section 8.3.1.

(b) **Record and restore C++ data structures.**
It has to be possible to record and restore C++ data structures. Especially, it has to be possible to record and restore classes including non-public members, structures and lists. Moreover, the *SSF* has to be able to work with "NULL"-pointers, e.g., the "NULL"-pointers shall be stored in the trace file and restored during the stubs execution.

(c) **Simulate the functional behavior.**
The *SSF* shall be able to restore the functionality of the component under study. Moreover, it has to be able to restore all recorded C++ data structures into the memory of the system under test. Additionally, it shall be able to create objects if they are not available in the system.

(d) **Simulate the functional behavior with appropriate performance.**
The *SSF* has to be able to restore the functionality in negligible time, which is at least faster than the execution of the original software function. This is necessary to optimize the runtime overhead. Hence, the performance parameters can be easily adjusted using the *performance simulation functions*. This requirement mainly applies if the *SSF* is used in the context of *DPS* performance measurements. In this case, the requirement has to be fulfilled.

Especially, the requirements to the system (Requirements 1a and 1b) as well as the Requirements 2b and 2c are important. Not reaching them renders the *SSF* unusable. Requirement 2d is mainly important in the context of *DPS* as this requirement enables the performance adjustments, which are necessary for the *DPS* approaches. This requirement may not be that important if the *SSF* approach is used in different scenarios. The Requirement 2a can only partly be fulfilled as stated in Section 8.3.1. The Requirement 1c is only suggested as it can significantly remove the overhead for applying the *DPS* framework in the performance evaluation study.

Moreover, there are some requirements to the C++ compiler [118]. The compiler shall be deterministic, e.g., the compiler has to produce an identical memory layout of two isomorphic classes. Whereas, this cannot be strictly guaranteed, it is very unlikely that a non-deterministic C++ compiler is standard-compliant [118]. The "g++" of the gnu compiler collection (GCC)[1], which is used for the evaluation in this paper, fulfills the requirements.

[1] see `gcc.gnu.org`

8.2 Methodology

The *DPS* methodology, identifies potential performance bottlenecks that are replaced by stubs to facilitate gain-oriented performance improvements. The functionality of the potential bottleneck (CUS) is recorded using the libSSF (see Section 8.3). The system's behavior with respect to the CUS is analyzed by replaying traces using the *SSF* with varying performance measures.

The overall process to replace the bottleneck by a *dynamic performance stub* consists of the following steps:

1. *Identify serialization objects:*
   The *SSF* is able to store and restore different states of the traced objects. Thus, it can be used to simulate the results of several algorithms. In some cases, it is necessary to use parts of the original functionality to improve the simulation results. Here, the content of the object may be stored and restored before and after executing the original software functions.

2. *Create serialization description:*
   In this step, the serialization description of the identified objects has to be created. This is simply done using the "GCC-XML" tool set [37].

3. *Create serialization specification:*
   The serialization specification is created. It contains a description to de- and serialize the objects which will be stubbed. The serialization objects (see Step 1) and their description (see Step 2) are processed by the *ssfheadgen* tool, which is part of the *libSSF* library, to generate a C++ header file that contains the serialization specification. This specification has been created automatically for many basic data types, as explained in Section 8.3.1, but, can also be easily extended by the developer to support object serialization. This header file will be included into the CUS.

4. *Record the state of the objects:*
   In this step, the component under study is adjusted to store the objects using the *libSSF*. Furthermore, the test cases that utilizes the functionality, which will be stubbed (see Step 1), have to be executed and the state of the objects have to be recorded into a trace file.

5. *Create functional software behavior:*
   The original functionality, which is a part of the CUS, is replaced by the *SSF*.

Usually, the objects, which will be restored are loaded before the SUT and CUS has been started. More details are described in Section 8.3.

6. *Test the instrumented CUS:*
   As the stub has been created in Step 5 the functionality of the stub has to be validated. The instrumented CUS is validated against the previously recorded behavior of the CUS that contained the original functionality. Hence, the measurements have to be redone and the results have to be validated. If the validation passes, the stubs can be used to do the performance study with the *DPS* framework.

This section has shown how a stub can be created using the *SSF*. The following section presents a possible realization called *libSSF*.

8.3 Realization

The implementation of the *SSF* is done in a library called *libSSF*. The library can be included into any C++ source code and provides the possibility to store the content of C++ data structures into a binary trace file. Moreover, the *libSSF* can also be used to read a from trace file to reconstruct the C++ data structures. Thus, the functional behavior of the component under study is also recreated. For this reason, the source code of the application will be parsed using the "GCC-XML" tool set [37], which generates an XML description of a C++ program from GCC's internal representation. Based upon this serialization description, *libSSF* generates an internal representation of the objects that will be stubbed. The following functionalities are provided by the *libSSF*. These are the general steps:

1. *Generate header file.*
   This file includes the serialization specification.

2. *Record functional behavior.*
   This functionality will be used to store the results of the software functionality of the CUS.

3. *Restore functional behavior.*
   This functionality will be used to simulate the software functionality of the CUS.

The listed items are described in more detail in the following.

8.3.1 Generate Header File (Serialization Specification)

A tool provided by the *libSSF*, called "ssfheadgen", parses the XML description of the "GCC-XML" tool. It extracts the serialization information and generates a C++-header file which includes the internal representation of the objects that will be replaced by the *SSF*.

This header file can be included into the C++ source code of the CUS and contains the de- and serialization specification of the objects. If others than the basic data structures, e.g., basic data types or fixed size arrays, have to be de- and serialized the developer has to adjust the header file to his needs. This has to be done manually as it is not always possible to determine the size of data associated with a pointer value.

Whenever possible, the header file already contains comments and suggestions to assist the developer in serializing the object, e.g., for pointers or arrays[2]. Moreover, the header file includes the original names, as used in the CUS source code, of the replaced objects, so that, the developer can reuse these names for convenience.

```
1 template <> void Stubfactory::serializeType(class array_class *
     ssfSaveObj) {
2   void *prt = ssfSaveObj;
3   struct ssfSave_array_class *ssfObject = (ssfSave_array_class *)
        ptr;
4   this->serializeArray(ssfObject->ac_i, numberOfElements);
5 }
```

Listing 8.1: Example: Serialize a Fixed Sized Integer Array, which is inside of a C++ Class

Listing 8.1 shows an example realization of the serialization of an array of integers ("ac_i"). The array is a private member of a class ("struct array_class"). This snipplet is used to deserialize as well as serialize the data values of the object.

The serializeType()-function from Line 1 will be called indirectly inside of the CUS. The provided parameter specifies a C++ class which shall be serialized and stored. In Line 2, a type cast of the object pointer to a void pointer is done. This is necessary for being able to furthermore cast the pointer to a "struct". In this case, the private or protected members of the provided class ("ssfSaveObj") can be accessed and, hence, stored. This is done in Line 4, where, the private member, which is a fixed size array in this example, will be copied into the trace file.

[2] In these cases a "stop criterion" has to be specified by the developer.

This example shows that private and protected members can be serialized. Other serialization functions are available to support the developer.

8.3.2 Record Functional Behavior (Binary Format)

The C++-header file, which has been generated by "ssfheadgen", will be included into the component under study. And, the software tests, which have been done to identify the serialization objects, have to be repeated. Now, the information of the objects will be stored in a trace file. This recording of the data structures is done using the function saveStateOfParam(), which will be included into the CUS. This function is provided by the *libSSF*. The declaration of the function can be seen in Listing 8.2.

```
template <class TYPE> void saveStateOfParam(const char *name,
    const char *type, TYPE *dataVar);
```

Listing 8.2: Stores a Data Structure (Class)

The following three parameters have to be passed to the saveStateOfParam()-function call:

1. "const char *name": This is the name used to store the object in the trace file, e.g., "conn".
2. "const char *type": This refers to the type and name of the object to be stored, e.g., "class Connection".
3. "TYPE *dataVar": This is a pointer to the data which will be stored, e.g., the value of the conn variable.

The three given examples in the list above can be interpreted as, store the value of the *conn* variable which has an object type *"class Connection"* into the trace file with the name *"conn"*. The function uses the parameters and stores the data structure as well as additional information into a binary trace file. The structure of a trace file entry is given and described below:

- *Test run*
  This is an internal reference counter starting from zero. The "test run" number can be used to summarize different stored variables into a combined run, e.g., if the value of a variable has to be stored before and after some modification within a single execution of the function.

- *Size of object name (byte)*
  The size of the object name is given in bytes including a “NULL”-termination character.

- *Name of the object*
  The name of the object which has been stored. It is usually the same name as the name of the object within the original source code and can be used for referencing the stored data.

- Size of the object type (byte)
  The size of the object type name is given in bytes including a “NULL” termination character.

- *Name of the object type*
  The name of the object type, e.g., “class Connection”, which means the data entry refers to a C++ class named “Connection”. Here, object type refers to any C++ data structure and can also be a basic data type such as an integer.

- *Additional information*
  The “additional information” flag, which is a bit field, is used to determine whether a fully initialized object has been stored or if a “NULL”-pointer has been passed. This information is stored in the first bit. The remaining bits of the bit field are unused. Hence, the first bit of “additional information” field is set to “0” if an initialized data structure has been stored. In this case, the following two additional data fields are stored in the trace file for this test run:

  – *Size of the stored data (byte)*
    This is the size of the serialized data in bytes.

  – *Stored data*
    The values of the data structure. The data have been serialized in advance and are successively ordered in the trace file.

The information are stored in a binary format for performance reasons. A decoded as well as semicolon separated example trace entry is given in Listing 8.3.

In this case, an object “conn” of the “class Connection” type has been stored. The values of the serialized private members are: “1”, “2”, “1” and “302845744”.

The *libSSF* provides a possibility to generate a trace file decoder for a dedicated trace file. This has been implemented to provide human-readable traces to the developer.

```
1;5;conn;17;class Connection;0;14;1;2;1;302845744;
```

Listing 8.3: Example: Decoded and Semicolon Separated Trace File Entry

8.3.3 Restore Functional Behavior (Deserialization)

The recorded values have to be recreated into the memory of the used C++ data structure. Hence, it is necessary to overwrite the values already stored in the memory of the object. To do this, three different cases has to be considered:

1. *The object as well as trace data are available.*
   In this case, the existing attributes of the object has to be overwritten as the object is already available in the system.

2. *The object does not exist but data are available.*
   The objects have to be created recursively and initialized using the values of the trace file. Moreover, the pointer to the object has to be returned to the system. This is possible as the delivered memory pointer has to be "NULL". In this case, no memory is associated with the original object. As there is no reference available, possible dangling pointers cannot occur.

3. *The Object does not exist and no data are available.*
   This case happens if an initialized object is not necessary, e.g., if the return value of a search algorithm does not find the item. I.e., the CUS returns a "NULL" value. In this case, the object pointer passed to the loadStateOfParam()-function of the *libSSF* (see Listing 8.4) has to be "NULL". Here, no memory will be allocated.

A fourth case, which can be thought of, is that an initialized object has been passed to the *libSSF* but no data are associated within this test run. In this case, a "NULL" pointer would have been returned by the *libSSF*, which will overwrite the original pointer value of the object. This is not allowed as it would cause a memory leak. Additionally, it is not possible to delete the associated object data as this could lead to a double free error. Hence, the developer has to care about this particular case, e.g., deleting the object and setting its pointer value to "NULL" before the "restore" function is called.

The restore functionality of the *libSSF* is realized by the loadStateOfParam()-function call. This function will be used to replace the software functionality of the CUS. The declaration of the function is given in Listing 8.4.

```
template <class TYPE> TYPE* loadStateOfParam(string dataVar, TYPE
    *dst);
```

Listing 8.4: Restore Functionality of the *libSSF*

The parameters passed to the *libSSF* are as follows:

- "string dataVar": This is the name of the object, which will be deserialized. Here, the same name as specified as the first parameter of Listing 8.2 is used, e.g., if "conn" is passed to the loadStateOfParam()-function, the with conn associated data will be returned.
- "TYPE *dst": This is a pointer to an object which will be overwritten by the values read from the trace file. As of, it is implemented as a template any type of the object can be deserialized and restored by the *libSSF*, e.g., the "class Connection" with an instance name "conn" can be used.

In the case that an object has to be created within the *libSSF*, the pointer value of the newly allocated memory will be returned to the component under study. Here, the original value of the pointer will be overwritten so that the allocated memory can be deleted inside of the original software.

The provided methodology and realization will be applied to a real world example which is presented in the following section.

8.4 Case Study

The *DPS* framework has been used to optimize several algorithms of a long term evolution (LTE [65]) telecommunication system.

This section describes the application of methodology and the newly developed *SSF* for the performance improvement study. The main contribution of this case study is to show that the software functionality of the CUS can be replaced by the *SSF*. This includes the following steps:

- The serialization specification is generated.
- The software functionality of the CUS can be recorded.
- The software functionality of the CUS can be replaced by the *SSF*. In this case, the SUT shall be fully functional for this particular test scenarios.

Last but not least, the case study shall provide some performance measurements to validate that the *libSSF* can be used in the context of the *DPS* framework that will be used to evaluate software performance optimization potentials.

8.4.1 Evaluation Environment

The measurements have been done in a host test environment. The used platform is a based on an Intel Xeon CPU, which is an IA-64 architecture.

The application has been built using the available build system of the company. This uses the “g++” of “GCC” (Version 3.4.3) for host test environment evaluations. The “-Os” compiler option has been used, which is basically a “-O2” but without optimization flags that increases the code size.

As the presented measurements have been done in a host test environment, the results can only be used for validation purposes of the *SSF* but not for performance test results.

The requirements to the software and test environment, as specified in Section 8.1, for using *DPS* are fully met. These are, in particular, a deterministic component under study as well as an automatic and reproducible test case execution environment.

8.4.2 Application of the Methodology

The system under test has a “ConnectionContainer” class which stores several connections of the type “class Connection”. The function “get(connID)” returns the connection specified by the connection identification (“connID”) which is an object of the “Connection” class. Moreover, it returns “NULL” if the connection does not exist in the “ConnectionContainer”. The connection class has basically four private members as can be seen in Listing 8.5.

Step 1 The “get(connID)” function has been identified as bottleneck and, hence, the “Connection” class has been chosen for serialization.

Steps 2 & 3 In the next step, the members of the “Connection” class are serialized using the “GCC-XML” tool set (Step 2). An example serialization output of the *ssfheadgen* (Step 3) is shown in Listing 8.6.

Here, only the first member “m_connectionId” is presented. The “GCC-XML”

```
class Connection
{
  ...
  private:
    TL3ConnectionId    m_connectionId;
    u16                m_streamId;
    TUeContextId       m_contextId;
    TAaSysComSicad     m_uecAddress;
  ...
}
```

Listing 8.5: Excerpt of the Class “Connection”

```
name='m_connectionId' id=_4096 type=_1501
  -> name='TL3ConnectionId' id=_1501 type=_1532
    -> name='u32' id=_1532 type=_73
      -> name='unsigned int' id=_73 type=
```

Listing 8.6: Example of Serialized Class Member

combined with the *ssfheadgen* tool identified the “m_connectionId” over four serialization steps as an unsigned integer.

As of Step 3, the serialization specification is written into a C++ header file. The first part of the file contains the serialized object, which is presented in Listing 8.7. As can be seen, the “Connection” class, which has been converted into a data structure, consists of four “private” members, which are basically integers.

The second part, which is the serialization code, is also included into the file. An extract is shown in Listing 8.8 for this case study.

```
struct ssfSave_Connection{
  unsigned int          m_connectionId;
  short unsigned int    m_streamId;
  unsigned int          m_contextId;
  unsigned int          m_uecAddress;
};
```

Listing 8.7: Serialized “Connection” Object

Here, the serializeType()-function in Line 1 is able to serialize a object of the “Connection” class. It calls internally several different “serialize” functions (Lines 4 - 7), which overloads the function from Line 1. The “serialize” functions from Lines 4 - 7 call internally a serializeAtom()-function, which is able to store and restore basic data types. The values of the variables are stored in their associated members of

```
1 template <> void Stubfactory::serializeType(class Connection *
    ssfSaveObj) {
2   void *ptr = ssfSaveObj;
3   struct ssfSave_Connection *ssfObject = (struct
      ssfSave_Connection *) ptr;
4   this->serializeType(&ssfObject->m_connectionId);
5   this->serializeType(&ssfObject->m_streamId);
6   this->serializeType(&ssfObject->m_contextId);
7   this->serializeType(&ssfObject->m_uecAddress);
8 }
```

Listing 8.8: Serialization Specification of the "Connection" Object

the data structure (see Listing 8.7). The "type casts" in Lines 2 and 3 are necessary to access the private members of the "Connection" class (see Section 8.3.1).

Step 4 Now as the setup has been finished, the measurements have to be repeated to store the software functionality of the component under study. The chosen test case is a functional test case which evaluates different use case scenarios. We only studied a small subset of the test case for the *libSSF*. In our context the test case includes 40 times calling the stubbed functionality ("get(connID)"-function). The test case includes the following use cases: "create new object", "reuse existing object" and "delete and create new object". A decoded excerpt of the recorded trace file is shown in Listing 8.9. Line 3 of the listing shows the recorded values of the private members of the "Connection" class for the second test run.

```
1 testrun:0; sizeObjectName:5; objectName:conn; sizeObjectType:17;
    objectType:class Connection; flag:1;
2 testrun:1; sizeObjectName:5; objectName:conn; sizeObjectType:17;
    objectType:class Connection; flag:0;
3  sizeOfData:14;
4   m_connectionId:1;
5   m_streamId:2;
6   m_contextId:1;
7   m_uecAddress:302845744;
```

Listing 8.9: Excerpt of a Decoded Trace File

Steps 5 & 6 In the last two steps, the stub has to be created using the restore functionality. Moreover, the proper working of the stub has to be validated.

The functionality which will be replaced by the stub, is removed from the CUS and replaced by the restore functionality of the *libSSF* in order to simulate the original functionality. Now, the test case, as used in Step 4, is executed and the results are validated. In the test environment the test case passed. In this case study, the *libSSF* were able to simulate the functional behavior of the component under study.

8.4.3 Performance Measurements

The concept of the *SSF* will be used within the *DPS* framework to simulate different performance behaviors of a software bottleneck. Hence, the time to restore the functional behavior of the component under study is critical.

A case study using the *DPS* for optimizing CPU bound systems has been presented in [125]. The focus was to optimize a CPU bottleneck. The case study in this section uses the measurement results of [125], but, interprets the results from a different point of view.

Here, the differences between the execution time of the component under study and the execution time for the *SSF* have been evaluated. In the case study of [125], a previous version of the *libSSF* has been used. However, the results of [125] can still be used for this evaluation as only smaller changes have been done.

The same test environment and software functionality, as described in [125] has been used. A description of the environment and software functionality can also be found in Section 8.4.1. The chosen test case started with a single database entry and ramped up to searching 400 database entries.

In Figure 8.1, the time behavior for searching an entry in the database (y-axis) depending on the amount of database entries (x-axis) is presented. The lower line (blue) shows the results for restoring the functional behavior of the search algorithm by using the *libSSF*. The upper line (red) depicts the original behavior of the component under study.

The new evaluation pointed out, the average time for restoring the functional behavior of the "get(connID)" function is 11 µs[3]. The according SCV is 0.0135. This factor indicates that, it takes approximately always 11 µs without significant variations to simulate the functionality independent of the amount of database entries.

In contrast, the original functionality to identify the connection identification

[3]The first message has been ignored to avoid side-effects that only occur for the first message ("first message effect").

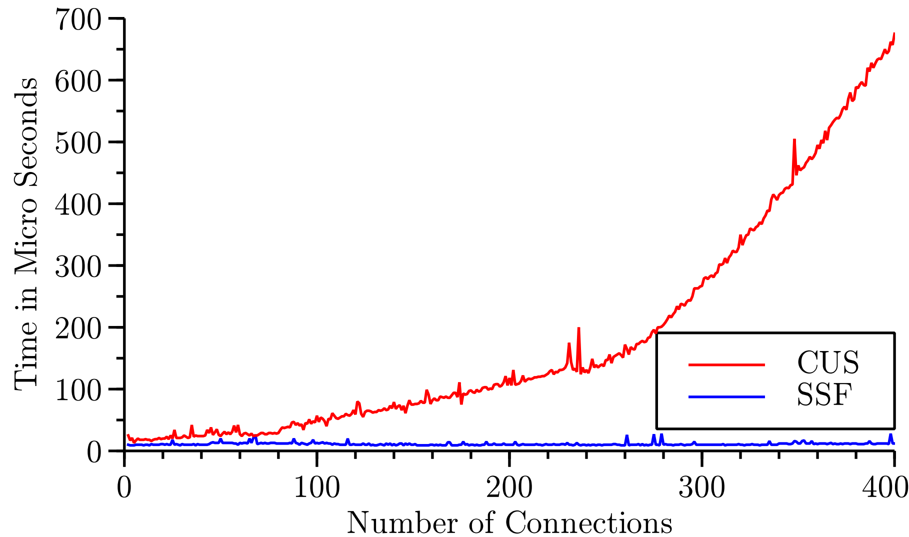


Figure 8.1: Compare the Times Between Original and Stubbed Software Functionality

number ("connID") took a minimum of 22 µs if only a single entry was in the database and about 675 µs if 400 entries has to be searched. The measured results show that time increases with the database size.

As can be seen, the *SSF* was even in the worst case as twice as fast. Due to this, the identification of the software optimization potential and the improvement of the bottleneck's time behavior were easy to be realized. Moreover, the methodology of using *CPU stubs* has been applied to the *system under test* in [125], successfully.

8.5 Discussion

The *SSF* can be used to store and recreate the functional behavior of software modules or functions of C++ applications. This is realized by the possibility to store and restore the values of C++ data types, e.g., data structures or classes including their private and protected members. Moreover, it is possible to use the *SSF* with applications which uses many different programming techniques, such as virtual or abstract classes, inheritance or polymorphism. The functionality is realized by a library called *libSSF* which can be included into the C++ source of the application.

Advantages Using the *libSSF* has several advantages for the developer. The main topics are:

- **Store and restore the software functionality**
  The *libSSF* can be used to record and restore the states of traced objects. Hence, it can be used to simulate software functionalities and algorithms, e.g., search-, sorting-, or calculation algorithms. Moreover, existing objects can be modified to the needs of the developer.

- **Mainly automatic header generation**
  The header, which is generated by the *ssfheadgen* tool, can be easily included into the source code of the application under study. Here, only some small modifications have to be done. Moreover, the *ssfheadgen* tool provides suggestions to support the developer by this task. This enables the developer to easily trace and evaluate the content of C++ data types.

- **Reuse object names**
  Data types can be stored into and read from the trace file reusing the same names as in the original source code. This significantly reduces the complexity to use the *libSSF* inside of the component under study.

- **Using data types multiple times**
  The same variables can be recorded multiple times, even within one single execution of the component under study. Moreover, several different data types can be combined into dedicated runs as well as many different runs can be combined. This provides high flexibility for clustering different runs and data types for a better abstract view on the stubbed components.

- **Readable values of the objects**
  The *libSSF* provides the possibility to decode the binary trace files into human readable trace files. Hence, the values of recorded data types can be used for evaluating the outcome of algorithms and, hence, as additional debugging possibility.

- **Only small adjustments to the system**
  To simulate the software functionality, only smaller adjustments to the component under study have to be done. For recording, only the library has to be included as well as the necessary function calls have to be added. For restoring, additionally, the original functionality has to be removed, e.g., by commenting.

Restrictions As often, there is a trade off between time and memory usage. If the library is used to restore the functionality of the software it will read the whole trace file into the memory while the initialization is done. Hence, it uses a lot of memory during the execution. Moreover, if a data type has been often recorded, each traced value is preloaded into the memory, e.g., if an integer has been stored ten times the *libSSF* will allocate ten times the size of the integer. This behavior has been chosen as the main focus has been on the execution time of the restore functionality. It can be changed with some smaller modifications to the library to only load the data when they are needed. This leads to a longer execution time, of course.

Another important restriction of the realization of the *SSF* is that some core features of the *libSSF* has been realized by kludges, e.g. a void pointer cast (see Section 8.3.1). Here, it cannot be guarantied that the approach works for different platforms or systems. Moreover, the usability of the *libSSF* might not be given in general.

Nevertheless, the usability of the approach has been presented in a real environment to simulate the functional behaviors of several different software modules.

Summary As can be seen, the methodology of the *SSF* as well as their implementation, realized by the *libSSF*, can be used to record and restore the software functionality. Moreover, the time measurements of the *libSSF* has shown that it can be used in the context of the *dynamic performance stubs* framework in this case.

This is an important contribution to the gain-oriented performance improvement framework *dynamic performance stubs*, as it allows to gauge the system-wide impact of a potential improvement before investing in the actual optimization of the algorithms that underly the functionality that has been simulated by the *dynamic performance stubs*. This informs decision making as to what bottlenecks should be prioritized and to what degree their optimization has a system wide impact.

The requirements on the system, which are 1a, 1b and 1c, as well as to the *simulated software functionality* (2b, 2c and 2d) as stated in Section 8.1 have been fully fulfilled with the *libSSF*. Finally, the Requirement 2a has been fully fulfilled in this particular case study, but, this cannot be applied in a general way as explained in Section 8.3.1. However, this does not lower the contribution as the *libSSF* supports the possibility to manually adjust the serialization functions. And, hence, provides a broad range for applying the *simulated software functionality* to software systems.

In the previous chapters the framework of the *dynamic performance stubs* as

well as different *performance simulation functions* has been presented. Within this chapter, the *simulated software functionality* has been evaluated. The *performance simulation functions* and the *simulated software functionality* can be combined to build *dynamic performance stubs.* The next chapter concludes the *dynamic performance stubs* approach as well as the *performance simulation functions* and the *simulated software functionality.* Moreover, it provides suggestions for future work to extend the overall approach of *dynamic performance stubs.*

Chapter 9

Validation of the Performance Simulation Functions

This chapter describes some measurements to validate the functional behavior of the performance simulation functions and the simulated software functionality. Therefore, it shortly describes the test environments as well as the test execution.

9.1 Test Environment

As a validation environment, the following equipment has been used to validate the measurements. It hosts a 2.8 GHz Intel Pentium 4 central processing unit with hyperthreading disabled. The operating system is a standard Linux running on a 2.6.22 kernel. The kernel was built tickless with the high resolution timers enabled. We ran the tests with executables generated by the GCC of version 4.2.1. In order to avoid unwanted optimizations by the compiler, optimization flags were not used for compiling the stub.

9.2 CPU Performance Simulation Functions

All measurements are done on an otherwise idle system, where only core processes are running. We also disabled the CPU frequency scaling feature in all our test runs. We have chosen the GNU compiler collection (GCC)[1] as compiler for our dynamic stubs.

We used a deterministic load if we needed to test our *DPS* in a situation where the system was not idle. This ensures the reproducibility of the results. Typical workloads are either a process, which uses a dedicated amount of the CPU or the process itself running multiple times. Despite that all examples presented in this paper are realized as global stubs, the results are also applicable to stubs in general.

9.2.1 Simulation of the Time

This section shows that the time can be simulated with high precision. Therefore, the methodology for calibrating the *CPU PSF* as described in Section 5.3 has been used. Our goals were to validate:

1. The simulation of different time intervals within a big range.
2. The accuracy of the simulated time interval.
3. The usability in different software development environments.

First of all, the “number of loops” has been determined for one second. This is the base for all measurements, which has been done. The main loop for each run can

[1] http://gcc.gnu.org

be seen in Listing 9.1, where "workInit" was initially set to the "number of loops" needed to simulate the highest simulated time value.

```
for ( i = 0 ; i < samples; i++)
{
    work = workInit * i;
    beforeTSC=readTSC();
    for ( j = 0; j < work; j++)
        ;
    deltaTSC[i] = readTSC() - beforeTSC;
    usleep(1000);
}
```

Listing 9.1: Simulation of a Duration

The graph in Figure 9.1 shows linear increasing of the "number of loops", which was initially set to simulate one second. The y-axis plots the needed time used inside of the application as TSC value. On the x-axis the actual number of samples can be seen. In Figure 9.1 each sample took 1ms.

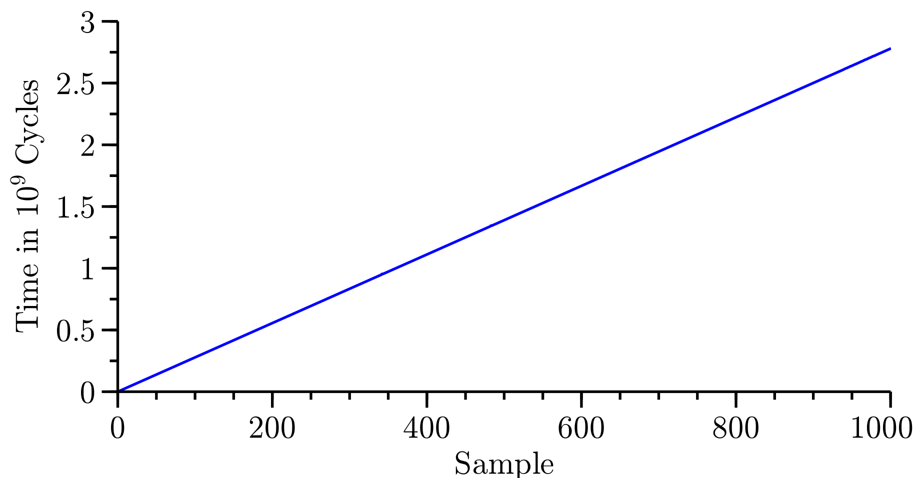


Figure 9.1: Simulation of Time Ranges

In order to prove the accuracy of the results, the overhead for measuring was subtracted from the measured TSC value. Additionally, the measured TSC value has been divided by the sample number and the value for zero loop iterations has been ignored. This results have been used to analyze the minimum, mean and maximum values and to calculate the SCV. The results of the evaluation of the values from Figure 9.1 are summarized in the second row of Table 9.1. Additionally, the table

shows further evaluation done as described above. Here the time ranges from 0 to 1µs and from 1 to 10 seconds.

range [s]	# loops	samples	min	mean	max	SCV
[0;1] µ	[0;398]	100	26.1	34.15	82.83	0.10794
[0;1]	[0;462356181]	1000	2777453	2779004	2793478	0.000000055
[0;10]	[0;4294963200]	1000	26861410	26879512	27023427	0.000000169

Table 9.1: Trustworthiness of the Time Simulation

Since we have shown that our approach of simulating time can be used to create *CPU Stubs*, we wanted to ensure that it can also used in different software development settings. Therefore, we firstly changed the compiler and secondly the operating system and architecture[2].

In Table 9.2 the results of this evaluation are shown. The first row is taken from Table 9.1. This line will be compared to measurements, where the binary has been built with the Intel C++ compiler (ICC)[3]. The results are almost the same and do not show any surprising values.

The values of the second test, which are presented in the last row of Table 9.2, cannot directly be compared to the other values. Here, an Intel Pentium M with 1.6GHz and Microsoft Windows XP Professional[4] has been used. The results show some slightly worse behavior, which can easily be explained by the change of the operating system. The used operating system has been running more "core"-processes and has not been optimized for this special task. Where else, the Linux Kernel has been explicitly built and optimized, no changes were applied to the kernel of Microsoft Windows XP Professional. Nevertheless, the values are still fine and our methodology can also be applied in this environment.

range [s]	# loops	samples	min	mean	max	SCV	setup
[0;1]	[0;462356181]	1000	2777453	2779004	2793478	0.000000055	Linux & GCC
[0;1]	[0;462819751]	1000	2777064	2778785	2793469	0.000000032	Linux & ICC
[0;1]	[0;225197530]	1000	1286750	1578846	1583827	0.0000518	Windows & GCC[5]

Table 9.2: Portability of the Simulation of Time

Conclusion We have experimentally proved that the time and, hence, the usage of the processor can be simulated by *system influencing CPU PSF*. Here, the values

[2] An Intel® IA-32 CPU has been used.

[3] Intel® and executed in the same environment. The ICC can be found at `www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm`

[4] Microsoft® Windows XP Professional®

[5] Intel® Pentium M (IA-32 Architecture)

for simulating time spans from several nano seconds to a couple of seconds and possibly more. Also the accuracy of the simulated time is sophisticated enough to use the *system influencing CPU PSF* for simulating the CPU behavior of processes. Additionally, a different software development setup do not highly influence our approach as proved above.

9.2.2 Open Loop

"Open Loop" in this context means that the stub should consume a dedicated amount of CPU usage for a definable time slice, e.g., 50% for the next 5 minutes. This can be used to simulate the performance behavior of a CPU bound bottleneck in order to replace it properly as part of the *DPS* framework. The open loop can be realised by a period with many sleeping and working faces by turns. So, no feedback loop is in place.

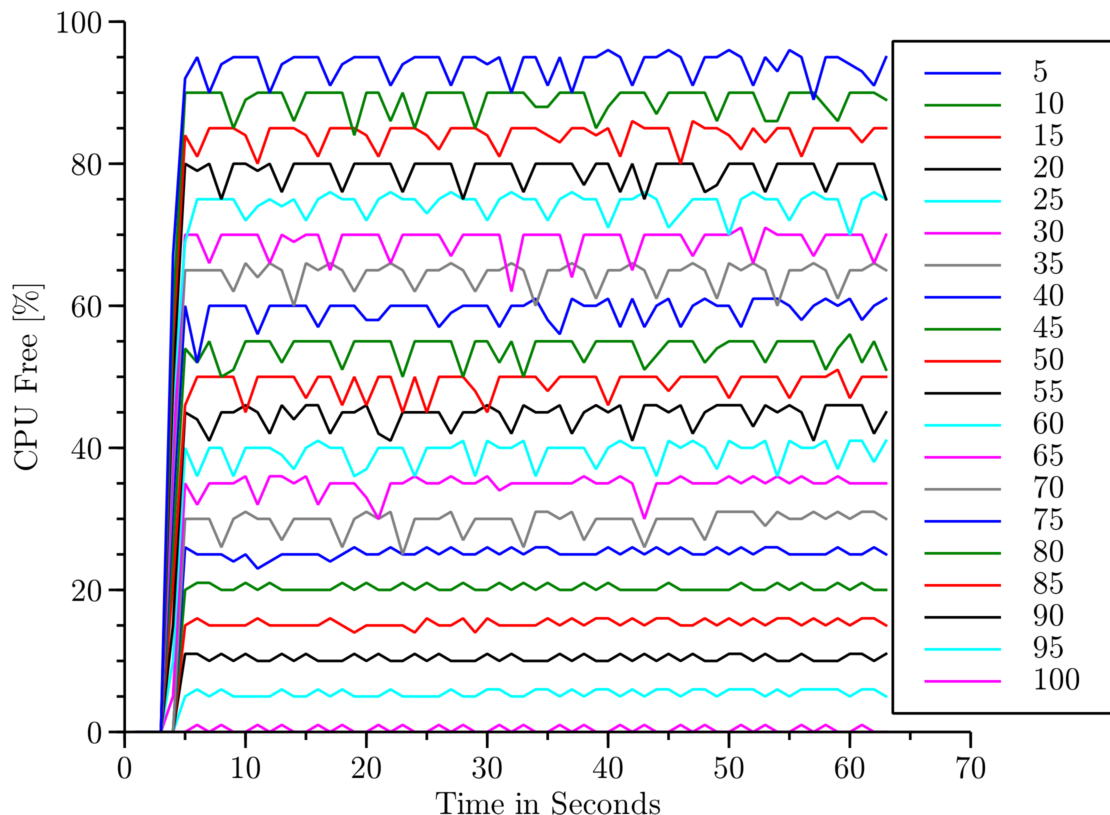


Figure 9.2: Global Open Loop CPU Stub Using a Dedicated Amount of Time

Figure 9.2 shows 20 single runs starting from 5% CPU utilisation with a 5% increase per step. The y-axis shows the idle value of the CPU and the x-axis provides the seconds since the test run was started. Each run took approximately 64

seconds where the first 4-5 seconds were used to setup the working and waiting periods including the calibration of the *CPU PSF* to the system. The graphs have been measured in the system using a standard performance measurement tool. The tracing interval was set to one second, which explains the two starting points of the simulation.

As can be seen in the Figure 9.2, the stub uses a dedicated amount of the CPU. Due to the open loop, it will not control the system utilization and, hence, it will not adjust it is values. The peaks, which can be seen in the graphs, can be explained by the work, which has to be done by the system.

This behavior can be used for a *local CPU Stub* to simulate a process, which periodically works and than waits for an event. Another possibility is to use it as a global stub to increase the CPU utilization or to transfer a system from non- to CPU bound.

The time used for the open loop was calibrated to 1 second. This means that, there were always "huge" blocks of "waiting" and "working", e.g., for 50% there is 0.5s working and 0.5s waiting.

This behavior mostly appears in a range around the middle percentages and is only hardly true for real applications.

For smaller percentages the behavior of "1 second" calibrated stubs can be taken for simulating "special" kind of applications, e.g., I/O bound, where the process shortly works and then waits for further input. For higher percentages the stub can simulate, e.g., number crunching applications. Here, large "working periods" are shortly interrupted by small "waiting periods".

We have also evaluated several approaches to scale down the "big blocks". Therefore, we have redone the measurements with smaller blocks of "waiting" and "working".

Scaling The size of the finest granularity depends on the smallest time, which can be simulated. As shown in Section 9.2.1, the smallest "working" time is bound close to the length of a couple of cycles. Despite the fine granularity of the "working" time, the smallest reliable "waiting" time for userspace applications is 1µ second. Thus, the "waiting" period is the bottleneck for scaling down big blocks into smaller blocks. Our approach is to take the 1µs as the total waiting time for each loop. This means the 1µs simulates the 100%-x%, where x means "supposed working percent". Therefore, the time needed for the "x working percent" has to be added to the 1µs. This approach works really well if the percentage is small. If the "working percent"

increases the additional time will take the lion's share of the total time, e.g., for 98% working the total time is close to 100 * the smallest possible (here 1µs).

Therefore, we also thought about another approach. This is the least common denominator which evaluates the smallest common denominator of 100% and the desired x% value. For the example above, the total time to simulate 98% comes down to 50 * the smallest possible time. Additionally, we also allowed some inaccuracy to the simulated x percentage value. Instead of only allowing exactly x%, we extended the value to some user given interval, e.g., to simulate 67% the value has to be between [66.5;67.5]. For that reason, our algorithm matches the smallest possible denominator to create a percentage, which is in that range. For example, our algorithm returns for the input of 67% a numerator of 2 and a denominator of 3. Compared to the least common denominator, the algorithm brings the total time down from 100 * smallest time to 3 * the smallest time.

For the simulation of different kinds of processes also multiples of the least common denominator can be used.

9.2.3 Closed Loop

The "closed loop" tries to adjust the CPU utilization to an user specified percentage by using a feedback loop. We experimentally implemented a time series analysis algorithm. It is based on a simple moving average with evaluation of the last five values of the original CPU load. The sample rate is set to 4Hz.

We used three different workload types for proving the concept. The first execution was done in an otherwise idle system and the second with a constant CPU utilization where the original utilization was below the supposed percentage. The results are similar as the results of the "open loop". This is nothing special and, hence, not further discussed.

The third evaluation was done with a variable CPU utilization. The original load signal (origload) is presented in Figure 9.3. The figure shows on the x-axis the iteration of the evaluation step, the y-axis shows the utilization of the CPU. The system load signal ranges from almost idle with some small peaks to shortly fully utilization and then varies around the to-be-adjusted value, which was in this case 50%. The figure also presents the controlled process variable, which is the second graph in the figure and called adjusted.

The real utilization was measured in the system and can be seen in Figure 9.4. The axes of this figure are the same as the figure above except of the sample rate,

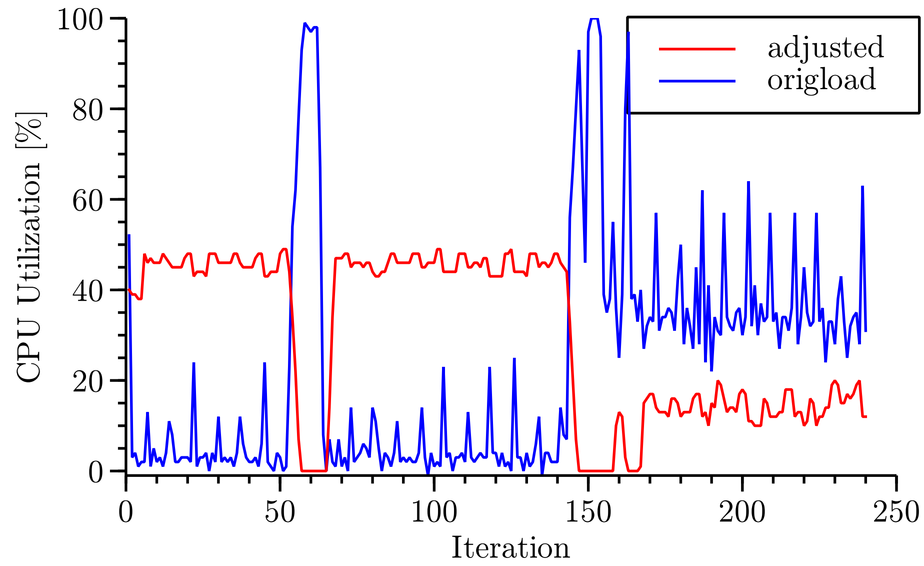


Figure 9.3: System Signal and Control Signal for the Closed Loop Algorithm

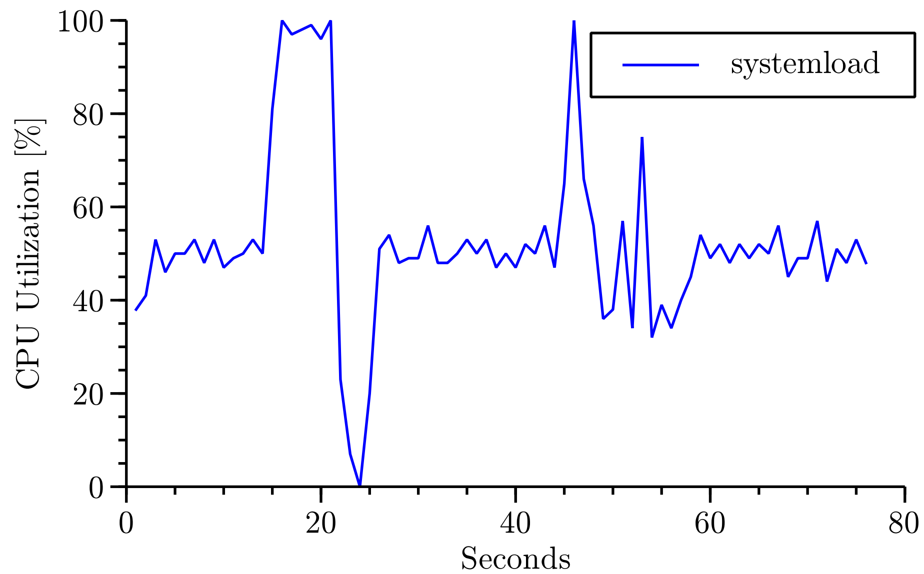


Figure 9.4: Total System Load Measured in the System while Closed Loop Algorithm was running

which was set to 1 second. The load in the system varies as supposed but the average is 52.40%, which is quite close to the predefined 50%.

Conclusion We have proved that the CPU utilization can be adjusted to a defined value as long as the original utilization of the CPU is lower than the supposed value.

This evaluation should only present the functional capability. A more theoretically and sophisticated methodology will be done as next steps. It will probably be based on the host load prediction system as described in [25, 26] and the Box-Jenkins methodology [40].

The closed loop can be used to adjust the utilization of the CPU to a defined value. This can be helpful for performance tests if the system should be tested under high load. Additionally, it can be used to deterministically create overload to validate the proper working and performance of overload routines under "real conditions" or to constantly influence other processes in the system.

9.3 Main Memory Performance Simulation Functions

The usability of the approach was validated twofold. First, an execution driven evaluation has been done. Thus, the execution time is validated using the TSC. Additionally, the amount of minor page faults is evaluated using the getrusage()-function. Moreover, a binary analysis has been done, where necessary.

The second stage of evaluation is simulation based. Hence, the valgrind tool suite, especially callgrind and massif, has been used. Callgrind is a callgraph and cache simulation tool. The results can be evaluated using KCachegrind. Massif[6] evaluates the stack and heap memory allocation behavior of processes by evaluating the allocation functions. More information on older versions of massif can be found in [96].

The single test runs are executed under the same test conditions. Each test has been executed several times in order to get statistically viable results. Hence, the minimum (min), average (mean), maximum (max) and squared coefficient of variation (SCV) has been used to validate the test results.

[6]Valgrind (massif) has been used in version 3.5.0.

Concept

This section evaluates different allocation and initialization possibilities as described in Section 3.3.2. Moreover, a new developed algorithm to use the allocated pages is described and evaluated. The validation- as well as the tracing environment, as described above, is used.

Next, a discussion of the behavior of the following functions is done: alloca(), malloc(), realloc(), memset() and distmemset(). The functions calloc() and memalign() are not discussed further. Because, calloc() basically combines malloc() and memset(), which are studied separately.

The test cases have been executed five times but only the evaluation of the third execution is displayed, unless otherwise mentioned. This has been done to simplify the reporting as all five tests show similar results.

Allocation Functions

An allocation function validates and ensures that the process will get enough memory as requested, if available. Hence, the heap allocation functions searches the already available heap memory for the amount of free space. If not enough space is available, it requests new memory from the system. This behavior is necessary as any heap memory can be freed during runtime, which leads to heap fragmentation. The stack memory allocation function basically only adjusts the stack pointer register.

Figure 9.5 compares the three mentioned allocation functions. The x-axis lists the number of the iteration done for allocating the 512 bytes of memory. The y-axis shows the time needed to execute the according function in terms of CPU cycles.

As can be seen in Table 9.3, an already prefetched alloca()-function call takes only 24 cycles on average. The first value of the test execution has been ignored because of the “first message effect”. The heap allocation needs 218 cycles for a malloc() respectively 279 cycles for a realloc()-function call. The time spent in the heap allocation shows a minimum value, as it differs in other scenarios, e.g., for a fragmented heap.

	min	mean	max	SCV
Alloca #3	24	24	27	0.00053
Malloc #3	210	218	259	0.00103
Realloc #3	259	279	903	0.14116

Table 9.3: Time Spent in an Allocation Function Call

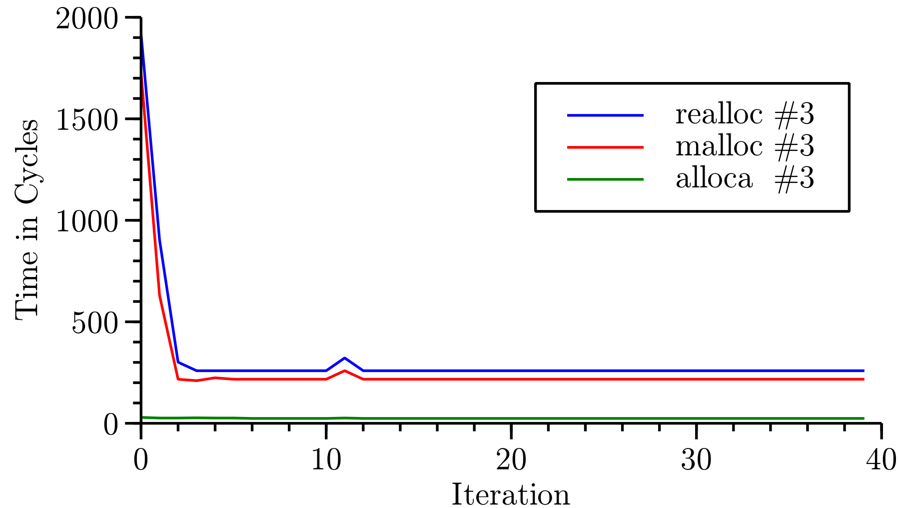


Figure 9.5: Time Spent in an Allocation Function Call

Additionally, the measurements have shown that no page faults are triggered by the allocation function, e.g., we measured only one minor page fault instead of the requested 20 pages with the alloca() function call. This is especially true for requesting many pages at a single time. The pages are only fetched as soon as the data will be used. Hence, a memory set function together with the allocation function is used to create page faults.

Memory Set Functions

As described above, allocation functions cannot solely be used to create higher amounts of page faults in the system. Thus, the memset()-function is used to initialize the memory. Hence, to create the desired number of page faults. As with memset() each memory position will be overwritten, we expected the time needed in the function to be directly linear to the amount of memory used. Moreover, changing the value of the initialization variable should not make any difference to the initialization time as the value should be available in one of the CPU registers. Besides of the timing evaluation, this estimation has been confirmed by an analysis of the binary.

The time spent in the memset()-function has been measured starting from initializing zero bytes to six pages. The pages have already been fetched and allocated in the process before measuring. Figure 9.6 displays the time needed to initialize one byte in cycles per byte on the y-axis. The x-axis shows the amount of allocated bytes. The blue graph (diamond) illustrates the time spent in memset() using al-

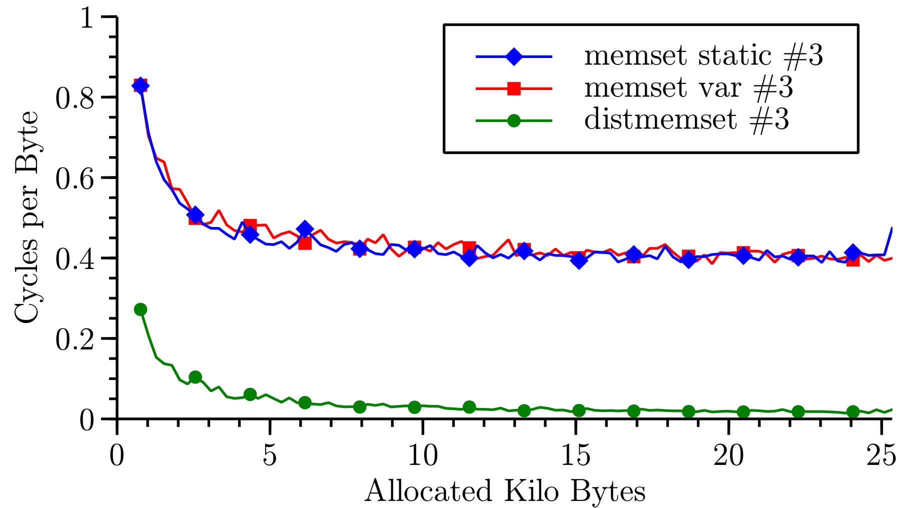


Figure 9.6: Different Memory Set Functions

ways the same initialization value "0" (memset static) and the red graph (squares) for changing the value using the actual allocation amount (memset var).

	min	mean	max	SCV
Mem. St. #3	0.389	0.434	0.828	0.02289
Mem. Var. #3	0.385	0.441	0.704	0.01168
Distmem. #3	0.014	0.038	0.208	0.48015

Table 9.4: Evaluates Different Memory Set Functions

Starting from approximately one page (4096 byte) the amount of cycles for setting a byte is almost constant. The average value is 0.434 cycles per byte for "memset static" and 0.441 for "memset variable", as can be seen in Table 9.4 Lines 2 and 3. Their respective SCVs are 0.02289 and 0.01168. The third test run results are displayed. The first three values of the tests have been ignored in Table 9.4 because of the initial overhead for small values.

It takes many cycles to set the memory using the memset()-function, e.g., ~10000 cycles for 6 pages with "memset static". As we only need to write into each page once to create the necessary page fault, we used an algorithm, which is similar to memset() but a distance between the initialized bytes can be specified. The algorithm is called distmemset(). The distance is set to pagesize in the test runs, so a time efficient generation of page faults has been achieved. The results are displayed in Figure 9.6 and Table 9.4.

The time used for initializing one byte in this test environment takes 0.038 cycles on average, which reduces the number of cycles to ~680 for six pages. We are aware

that our algorithm is much slower than memset() if it is used with a distance of one byte. But, this does not matter as we only want to set one byte per page.

Another advantage of distmemset() compared to memset(), with our test conditions, is that, we only want to simulate the heap and stack behavior with the *mm stubs*. The rest of the system should not be significantly influenced by the stubs, e.g., the *mm stubs* should not create many cache events. As memset() sets every memory position a lot of cache events, e.g., level one cache misses, can be seen. The same execution setup has been used to measure the total amount of cache write misses. Callgrind shows 61339 for "memset static" compared to 1143 for distmemset().

Time to Serve a Page Fault

The following test has been executed using the malloc() and distmemset()-functions. The time measured includes both function calls. Here, the time needed to allocate and load multiple pages to a process has been evaluated. The process starts using zero to 9900 pages with an increment size of 100 pages.

min	mean	max	SCV
8068	8628	8757	0.00023

Table 9.5: Time Needed to Allocate and Load a Page

As can be seen in Table 9.5, the average time to load and set a single page is 8628 cycles. In this case, the malloc() and distmemset()-functions have only small influences. The SCV reflects that the average value has only small variations. The first value, which is "allocate zero pages", is not shown and evaluated in Table 9.5.

As described in Section 3.3.2, memory that has been freed is not directly returned to the system but stored in a malloc pool. Hence, the process can reuse the memory if needed without requesting a new page from the system. The behavior of the malloc pool has been experimentally evaluated.

Malloc Pool

Figure 9.7 shows on the x-axis the number of each test run. The allocated size starts from zero to 114 pages with an increment size of six pages (20 test runs). The number of pages is listed on the y-axis. The blue graph (diamonds) shows the amount of allocation memory in pages. The red line (circles) displays the number of minor page faults created by the process. As can be seen between test run two and twelve, much fewer minor page faults happen as pages are being requested. In this

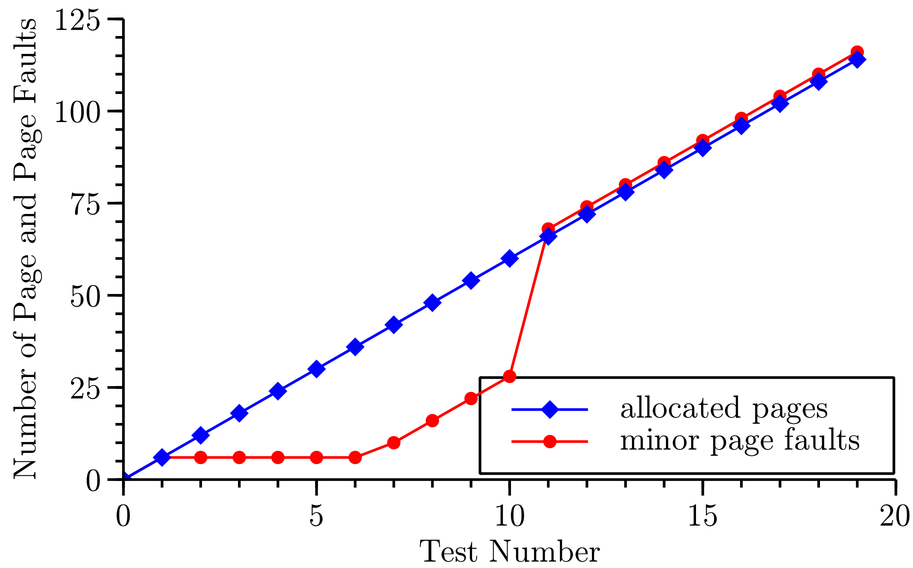


Figure 9.7: Supposed and Measured Page Faults

case, the pages from previous test runs are still available for the execution. Above 60 pages, which is ∼250000 bytes, the memory will be returned to the system.

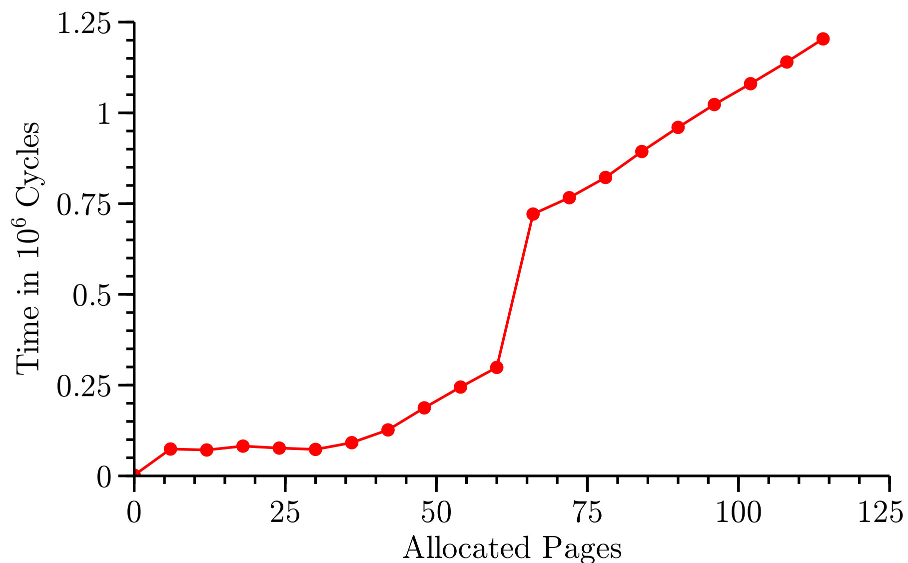


Figure 9.8: Time Influence of the Malloc Pool

The influence of the malloc pool can also be nicely displayed by the time needed to allocate and load a single page. This is shown in Figure 9.8, which has been taken from the test execution as described above. Here, only the time spent to answer the request is depicted on the x-axis.

Conclusion In this subsection different heap and stack allocation scenarios have been evaluated. As shown, each page has to be used at least once to really allocate the page to the process. Moreover, the time for different functions has been determined and the usability to simulate the memory behavior of applications has been shown. Hence, the functions provided can be used to create *mm stubs*.

9.4 Data Cache Memory Performance Simulation Functions

This section validates the *dcm PSF*. It is split in two different subsections. First, the access behavior of the algorithm itself is executed. Here, the algorithm is validated to show that it is working as described in Chapter 7. The second validation evaluates whether the algorithm can be used to deterministically create a particular amounts of cache set references (CSR).

9.4.1 Validate Access Behavior

Within this section, the different parameters of the algorithm are evaluated. Hence, the access behavior to create the supposed *CSR* is studied. To validate the access behavior, the algorithm has been annotated to show the accessed memory location in the array. For evaluation reasons, the following configurations have been used.

The cache for testing is configured as a 2-way set associative cache with a two byte cache line size and four sets. This leads to a cache size of 16 bytes. In Table 9.6 the cache configuration is depicted.

Cache Line	Assoc	Sets	Size
2	2	4	16

Table 9.6: Cache Configuration for Validating the Access Behavior

In Table 9.7 the basic configuration for the different test cases can be seen. Each row presents an excerpt of the parameters, which were written to the “datafile.h”.

The *cache* parameter was configured to use the cache as described in Table 9.6. The *accesses* and the *arraysize_max* parameters were set to 24. As in this validation neither the access *direction* nor the *sleeptime* parameter is evaluated. Their values have been set to *read* resp. to zero.

Test Case	Accessstride	Sets	Setstride
1	8	1	1
2	8	2	1
3	8	2	2
4	8	4	1
5	16	1	1

Table 9.7: Basic Configuration for the Validation of the Access Behavior

These configurations were chosen to validate the different configuration possibilities of the *dcm PSF*. Basically, the algorithm uses a single set of the cache in the first test case. Furthermore, the algorithm uses two different sets with different set strides (Test Cases 2 & 3). In Test Case 4 each set of the cache is used. Finally, Test Case 5 is similar to Test Case 1 but, here, the *accessstride* parameter has been varied to validate the usability of this parameter.

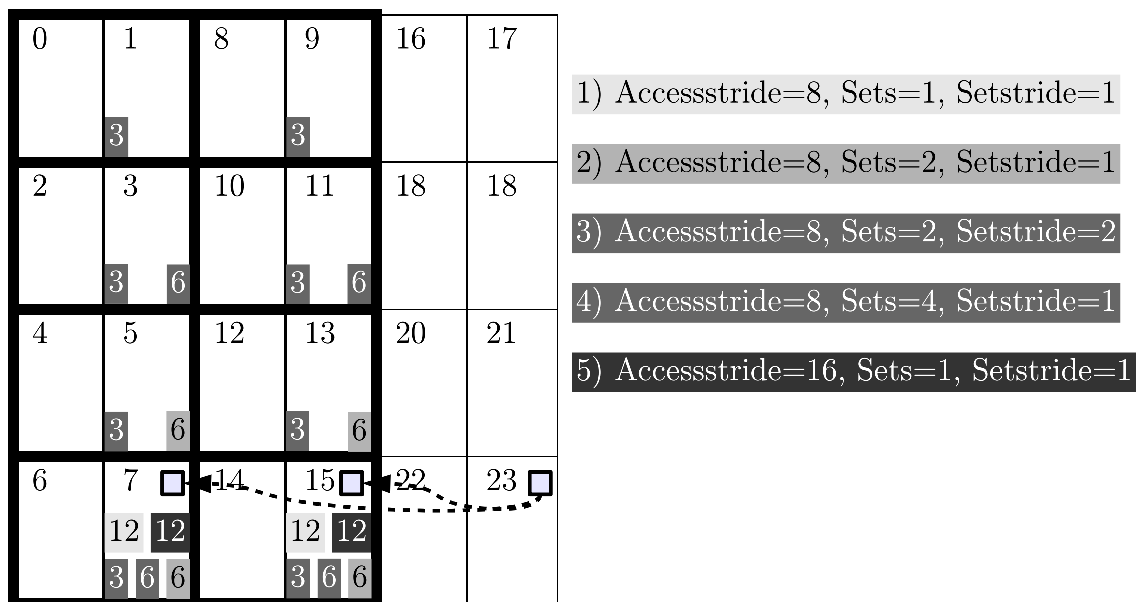


Figure 9.9: Validate Access Behavior of the *Data Cache Memory PSF*

In Figure 9.9 the access behavior of the algorithm is depicted. Each field presents a single byte in the array including the memory location. The fields surrounded by the bold lines present cache lines. The outer bold line presents the cache itself. As can be seen, the cache is 2-way set associative with four sets and two bytes per cache line. The total size of the cache is 16 bytes.

An access to a memory location "outside" of the cache is mapped to a cache line. An example is presented in Field 23. Typically, the cache line is determined

by the memory location modulo the cache size, e.g., 23%16=7. Because of the fully associative cache architecture within a single cache set (see Section 3.4.2), it cannot be guaranteed to be stored in this particular cache line. So, the data will be either stored in position 7 or 15 depending on the replacement policy.

The different test cases access different locations in the cache. These locations as well as the amount of their number of accesses are presented by the gray-shaded boxes including a number. The distribution of the accesses is based on a deterministic replacement policy, e.g., LRU (see Section 7.2.5). Different colors have been used for the different test cases as described above.

As an example, Test Case 4 uses four different sets. Here, each cache line is used three times. The locations as well as the number of accesses have been determined by the output of the annotated algorithm. As can be seen, the different test case configurations access different locations in the cache. Particularly, by comparing Test Case 2 & 3 the influence of the *setstride* parameter can be seen. In the first test case, the Sets 2 & 3 are used. In the second, 1 & 3 are used.

The difference between Test Cases 1 & 5 cannot be presented in the figure as the same cache lines are used by the algorithm. An analysis of the output has shown that Test Case 1 uses the array's memory locations 23, 15 & 7 eight times each. Test Case 5, however, uses the locations 23 & 7 twelve times each. Hence, the algorithm is working as expected for different *accessstride* parameters.

Conclusion This section evaluated the memory accesses behavior of the *dcm PSF*. Hence, the different parameters, i.e., *accessstride*, *sets* and *setstride* have been used for validation. Moreover, this section provides an overview on the influences of the different parameters to the access behavior. It has been shown that the algorithm works as expected. Thus, it can be used to create *CSR*.

9.4.2 Validate Cache Set References

In this section, the usability of the algorithm to deterministically create *CSR* is discussed. In the following, an overview of the caching architecture of the test environment is provided. Afterwards, the validation of the *CSR* is done for the various cache levels and cache access types.

Test Environment

The test environment of Section 9.1 has been used. This section shortly presents the caching architecture. More details about the mentioned characteristics can be found in Section 3.4.2.

The CPU has two different cache levels (L1 & L2). These levels are combined as non-inclusive and use a write-through policy. Each cache level is realized as an 8-way set associative cache with a cache line size of 64 bytes. The pseudo least recently used (PLRU, see Section 3.4.2) replacement policy is applied. A write buffer is associated with the L1 cache to avoid the occurrence of write misses in the L2 cache. A victim buffer is not available in the CPU; but, the out-of-order execution capabilities are included.

In the following, the particular characteristics of the two caches are depicted:

- The L1 cache is split into a data cache and a trace cache, which is an instruction cache storing micro operations (µops). The size of the level one data cache is 16 kB.
- The L2 cache is an unified sector cache to store data and instructions. Each cache line (sector) is split into two subsectors. The L2 cache has a size of 1MB.

The CPU provides several hardware counters to trace cache events. All available counters for measuring cache events are listed in the following:

- L2 read hit events.
- L2 read miss events.
- L2 writeback lookup misses.

Other cache events such as level one hits can not be recorded due to the lack of additional counters.

Register and Level One Hit

Within this subsection, a comparison between register and L1 cache hits is evaluated. As there are no hardware counters for the registers and L1 cache, time measurements have been used. The *dcm PSF* algorithm as described in Section 7.2 was applied to create the register accesses and L1 cache hits.

The configuration of the test cases are provided in Table 9.8. The evaluation has been done for read and write accesses. Each test case has been executed five

	Array Size	Access Stride	Sets	Set Stride
Register	1	0	1	0
L1 Cache Hit	64	1	1	0

Table 9.8: Configuration for Registers and L1 Cache Hits

times in order to get an estimation of the reproducibility of the *CSR*. The number of accesses for each events range from 100000 to 1000000 with a step size of 10000. The times were measured before the Lines 14 and 28 in Listing 7.4 using the TSC.

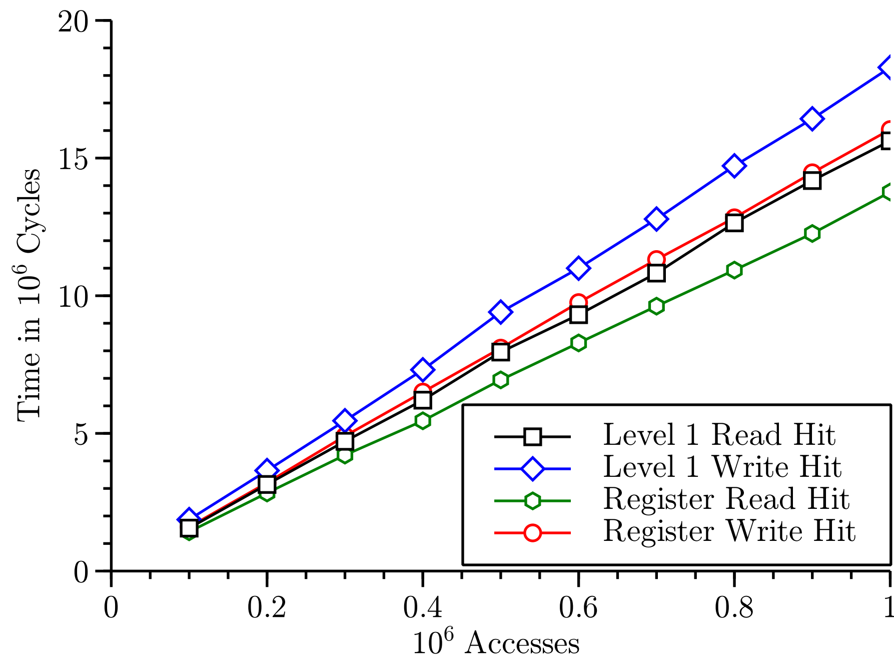


Figure 9.10: Time Behavior of Register and L1 Cache Hits

The results of the evaluation are presented in Figure 9.10. The number of accesses is shown an the x-axis and the time is presented in cycles on the y-axis. As can be seen, the time of creating the *CSR* is linearly growing with the number of accesses. Hence, a specified amount of *CSR* can be deterministically simulated with the *dcm PSF* for register and L1 data cache hit read and write events.

	Register (SCV)	Level One (SCV)
Read Hit	13.9 (0.00026)	15.7 (0.00008)
Write Hit	16.1 (0.00005)	18.4 (0.00013)

Table 9.9: Time Evaluation of the Registers and L1 Cache Hits

The average values and SCV for creating one register or L1 data cache event

is presented in Table 9.9. The values have been calculated using the results from Figure 9.10.

As can be seen in the table, the time for one iteration for the register read access is approximately 13.9 cycles. If the algorithm is configured to create a L1 data cache read hit, the time is about 15.7 cycles on average. Hence, the difference is 1.8 cycles, which is close to the time to access the L1 data cache for a read hit event (see Listing 7.6). The same can be seen for the write events. Here, the time for a register write hit is approximately 16.1 cycles. Hence, a register write access takes about 2.2 cycles longer than a read access. Moreover, the time difference between a L1 data cache write hit and a register write access is approximately 2.3 cycles, which is little more than the L1 cache access time. By comparing the read and write values, it can be seen that a write value always takes about 2.5 cycles longer than a read access.

The SCV values have been calculated based on the average for all different measurements per *CSR* type. The time spent to execute one event is equal for all different amounts of accesses within small variations.

To summarize, the *dcm PSF* can be used to create L1 data cache hits for read and write events. Moreover, it is possible to specify the number of *CSR*. Thus, an amount of *CSR* can be deterministically created for the level one data cache hit events, and, the number of accesses can be specified.

Level One Miss / Level Two Hit

This section evaluates the L1 data cache miss / L2 unified cache hit read and write events[7]. The read events have been evaluated using oprofile and time measurements. The write event evaluation has been done using time measurements as the appropriate hardware counters are not available. These counters are not available because of the caching architecture. Here, a write buffer is associated with the L1 data cache. The write-through policy of the L1 data cache updates the write buffer if a L1 data cache write miss occurs. The write buffer is then used to sync the data with the L2 cache.

	Cache	Access Stride	Set Stride
Read / Write	L1D	2048	1

Table 9.10: Configuration for L1 Misses / L2 Hits for Read and Write Access

[7] This is referred as “L1 data cache misses” in the following.

The configurations of the test cases are presented in Table 9.10. The maximum array size is set to the array size. The array size and number of sets were varied as presented in the figures and tables below. Each test case has been executed 25 times for the access' numbers 100000, 200000 and 300000.

Read Accesses (Oprofile Measurements)

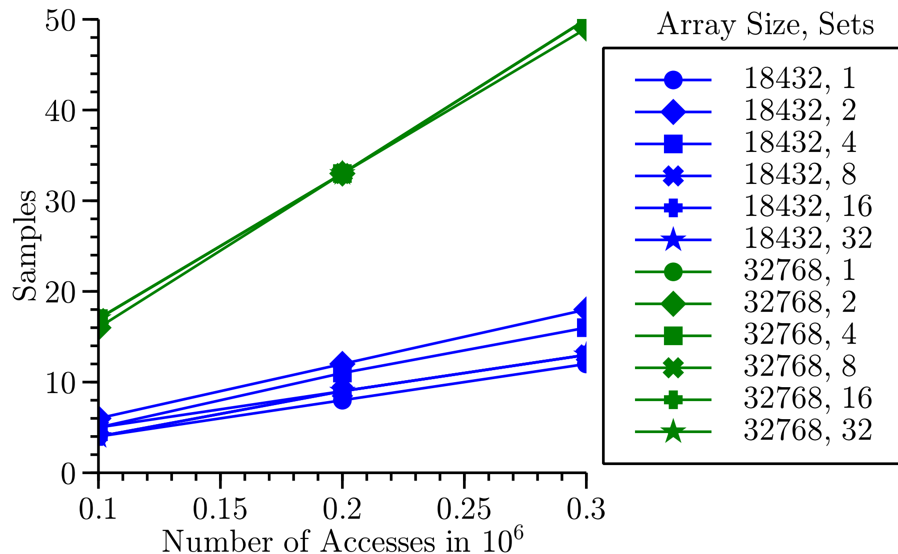


Figure 9.11: Samples of L1 Read Misses / L2 Read Hits for Different Array Sizes and Set Values

In Figure 9.11 the number of samples taken with oprofile is presented on the y-axis. On the x-axis the number of accesses is shown. The sets and the array size values have been varied as presented in the figure. Due to the sampling based measurement variations especially for small sample numbers can occur. The results, presented in the figure, show a deterministic behavior. The lower graphs presented in blue provide the number of oprofile samples for different set values for an array size of 18432 bytes, which is calculated on the assumption of the optimal array size as in Equation 7.1. The green graphs (higher graphs) present the oprofile samples of an array size 32kB, which is the doubled cache size.

As can be seen by the blue graphs, the *dcm PSF* does not create the necessary *CSR* for the optimum array size as described in Equation 7.1. But, for higher array sizes (green graphs) the *dcm PSF* can be used to create the necessary amount of *CSR*. Hence, two conclusion can be drawn from the figure. First, by comparing the blue and green graphs, it can be seen that the success rate to generate the cache

events depends on the array size. Second, for small array sizes the success rate depends additionally on the number of sets. This can be seen by studying the blue graphs. Additionally, results for higher array size values have also been evaluated but not presented as they are as expected (green graphs).

Oprofile was configured to create one sample after 6000 L1 cache read miss events. In the following, an evaluation of the measured and supposed *CSR* is done. This evaluation is based on the average values for the different access numbers over all sets. The array size was set to 2 times cache size (32768 bytes).

Accesses	Samples (avg)	Total Events (avg)	Success Rate (avg)
100000	16.83	101000	1.01
200000	33.00	198000	0.99
300000	9.57	299000	1.00

Table 9.11: Evaluation of Accesses and Samples

In Table 9.11 the success rate for generating L1 data cache read misses is shown. The first column presents the number of accesses. In the second column the average number of samples per access is presented. The third column depicts the number of events calculated by the number of samples (Column 2) multiplied by the oprofile sample event value (6000). Finally, the average of the success rate (Column 4) is evaluated by dividing the value “total events” (Column 3) by the number of accesses (Column 1). As can be seen in Column 4, the average success rate is approximately 1. This means that a predefined value of *CSR* can be deterministically created in the system for different sets values.

Read Accesses (Time Measurements)

In Figure 9.12 the time behavior for the L1 data cache read miss events is presented. The same configurations as described above have been used. The time (cycles) spent to execute the *CSR* is shown on the y-axis and the number of accesses is on the x-axis.

As can be seen, the time linearly increases with the number of accesses. Moreover, the results show that the algorithm has a deterministic cache access behavior. This is also applicable for different numbers of *sets*. Hence, the algorithm works as supposed for L1 data cache read misses / L2 cache read hits.

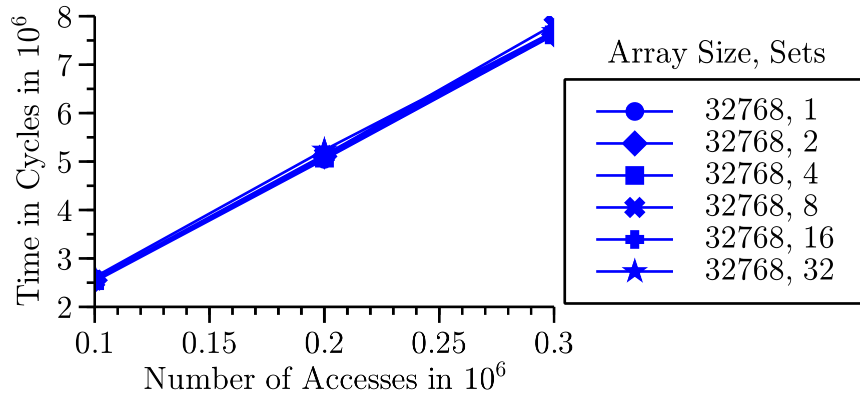


Figure 9.12: Time Behavior of L1 Read Misses / L2 Read Hits

Write Accesses (Time Measurements)

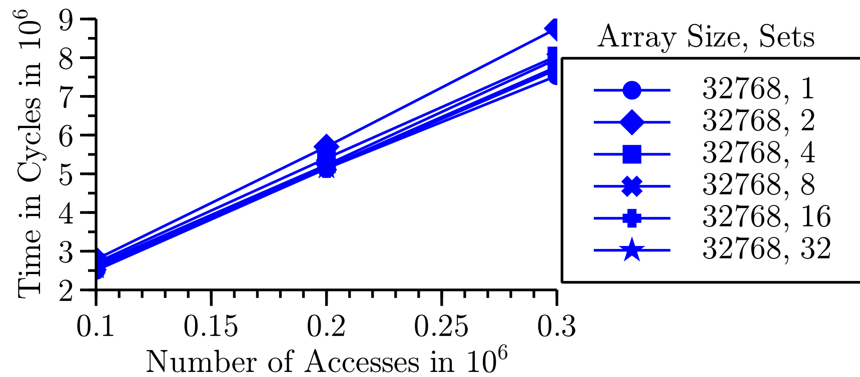


Figure 9.13: Time Behavior of L1 Write Misses

Figure 9.13 presents the time behavior for L1 data cache write misses. The x-axis shows the number of accesses and the y-axis provides the time in cycles. The test case configuration as described above has been used and the sets value has been varied according to the figure. Each presented result is the average of 25 independent test runs.

As can be seen in the graphs, the time linearly increases with the number of accesses. There are only some small variations, which do not influence the overall result significantly. Hence, the algorithm works as supposed for L1 data cache write misses.

Comparison of Read and Write Accesses (Time Measurements)

In Table 9.12 the average time for one iteration of the *dcm PSF* is presented for the previously shown read and write measurements. The evaluation has been done for different set values and an array size of 32786 bytes. The results show a deterministic access time for the set value configuration.

	1	2	4	8	16	32
Read	25.3	25.5	25.4	25.8	25.7	26.0
Write	25.3	28.6	26.9	26.4	25.6	25.9

Table 9.12: Time Comparision for Different Set Values

An evaluation of the time for read and write L1 data cache misses is provided in Table 9.13. Here, the values of Table 9.12 have been used to calculate the average access time. In the second row, the SCV values for this evaluation is provided to see the influences of the set value to the overall execution time.

	Read	Write
Cycles	25.6	26.4
SCV	0.00010	0.00201

Table 9.13: Time Evaluation for Read and Write L1 Data Cache Misses

An iteration through the loop to create an L1 data cache read miss takes on average 25.6 cycles. A register read hit takes approximately 13.9 cycles (see Table 9.9). The difference between those two accesses is about 11.7 cycles. This value is a little less as the L2 access time, which is 18 cycles (see Listing 7.6). The variation in time can be explained by the used array size. The algorithm takes the if condition in Line 23 of Listing 7.4 more seldom when looping through bigger arrays. The difference between a read and a write access is around 1 cycle.

To summarize, this subsection evaluated the L1 data cache miss / L2 cache hit read and write events. The results have shown that a predefined amount of *CSR* can be deterministically created. The evaluation is mainly based on time measurements as the appropriate hardware counters are not fully available on the system because of the cache architecture. The results are satisfying and also match other evaluations as well as the specification of the CPU.

Level Two Read Miss

In this section the generation of L2 cache read miss events is evaluated. Only the read misses are studied because a write buffer is used to avoid L2 write misses in the test environment. As described in Section 4.4 the *dcm PSF* are only designed to recreate the cache access behavior of processes for data and unified caches.

The test case configuration for the evaluation presented in this section is summarized in Table 9.14. Parameters, which are not presented in the table have been varied and are depicted in the figures and tables used for the evaluation. The array size and maximum array size are equal. The axis in the figures are the same as described in the sections above. The test cases have been executed 25 times to get a statistical distribution. The values presented are the average values of the test runs. Additionally, a SCV has been presented to show the deterministic behavior of the algorithm.

Cache	Access Stride	Set Stride
L2U	2048	1

Table 9.14: Configuration for L2 Read Misses

Read Accesses (Oprofile Measurements)

As shown in the subsection above, the success rate for the cache event generation varies on the parameters *sets* and *arraysize*. In the following, these two parameters are studied independently for the generation of L2 cache read misses.

Influence of the Sets Parameter

In Figure 9.14 the influence of the *sets* parameter to the simulation results is depicted. The array size has been set to 16777216 bytes, which is 16 times the L2 cache size. For smaller values of array size only few oprofile samples can be measured. Beside the deterministic measurement of these samples, the main focus of the *dcm PSF* is to generate an adjustable amount of cache events with only few side effects for the other cache events.

As can be seen in the figure, the number of measured oprofile samples depends on the *sets* parameter. Whereas, even in the best case, which is $sets = 2048$, the success rate for generating L2 read misses with an array size of 16777216 bytes is not equal to one. However, this array size value has been chosen as the influence of the

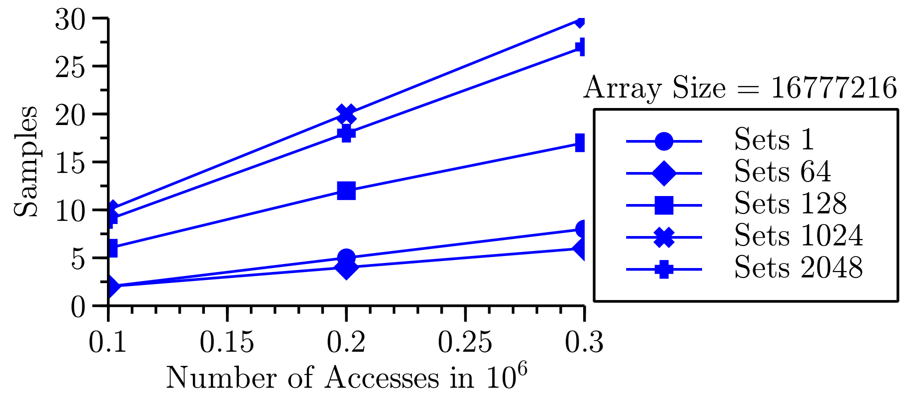


Figure 9.14: Influence of the Sets Parameter (excerpt)

sets parameter to the simulation results can nicely be presented. The figure only presents few different set values for clarification. The parameter has been varied from 1 to 2048 by doubling its value. The other results are as expected and, hence, are not further presented.

Influence of the Arraysize Parameter

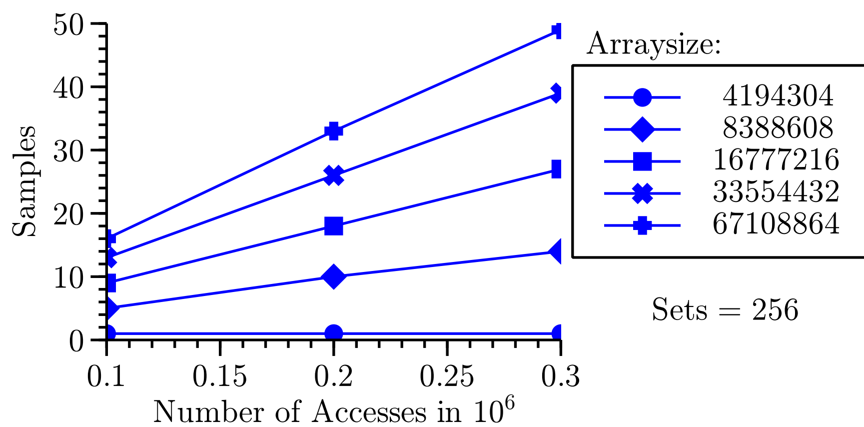


Figure 9.15: Influence of the Arraysize Parameter (excerpt)

Figure 9.15 presents the influence of the *arraysize* parameter. The *sets* parameter has been set to 256. The array size value has been varied from 2 to 128 times by doubling the array size. Additionally, a factor of 1.125 has been used as this presents the minimal array size as evaluated in Equation 7.1. In the figure only an excerpt of the different array sizes is depicted to increase the readability of the graph. The other results of the measurements, which are not presented, do show expected behaviors.

As can be seen, the success rate for the generation of L2 read misses strongly depends on the array size. Additionally, the minimal array size is 67108864 bytes to create the specified amount of cache events for a $sets = 256$ value. From further evaluations, the same array size can be used for a *sets* value equals one.

Deterministic Generation of L2 Read Miss Events

Starting from an array size of 67108864 bytes, the L2 read miss events can be deterministically created for any *sets* value. Hence, this is considered as the minimum array size.

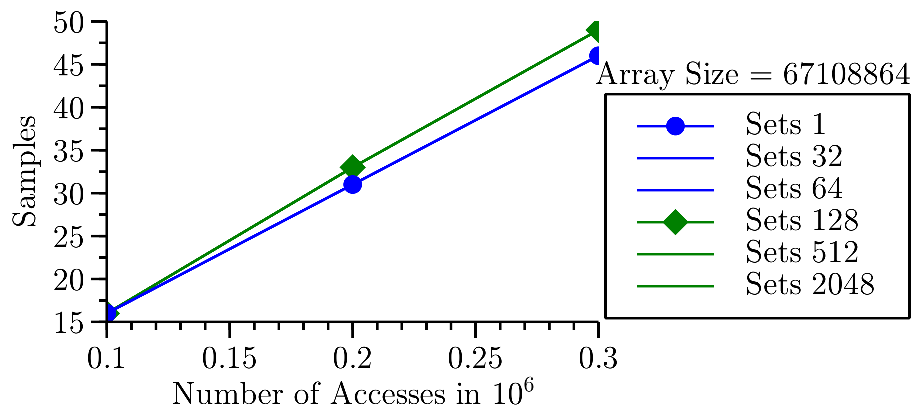


Figure 9.16: Samples of the L2 Read Misses for different Set Values

This is presented in Figure 9.16. Here, for any *sets* value the success rate for creating L2 read misses is approximately equals one. The measurement results for the blue and green graphs are the same.

Overhead of L1 Read Miss Events

Here, the number of generated L1 read miss events is studied. This can be considered as the overhead, which is generated in the L1 cache for creating L2 read misses.

In Figure 9.17 the L1 read miss oprofile counter has been used to measure the number of L1 read miss events for a generation of the L2 read miss events. The same configuration as described in Table 9.14 has been used. Usually, if a L2 read miss is caused a L1 read miss happened as well because the data has to be transfered into the lower levels, too[8].

[8]This is applicable in any inclusive cache architecture (see Section 3.4.2).

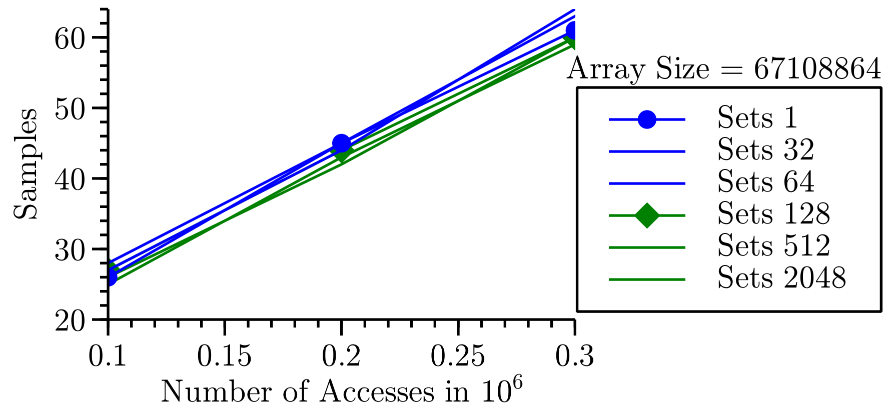


Figure 9.17: Generation of L1 Read Miss Events for Creating L2 Read Miss Events

As can be seen in the figure, a constant overhead of L1 read misses is created when generating L2 read misses. However, the overhead is deterministic for the different *sets* values as well as for the number of accesses. From further measurements, it can be seen that the overhead depends on the array size used for the generation of L2 read misses. For smaller array sizes the overhead is reduced. The correlation between the overhead and the array size is evaluated within the *CF*. Especially for simulations using the whole cache, this overhead can be reduced as the array size can be reduced. In this case, the *sets* parameter is set to its maximum.

Read Accesses (Time Measurements)

In this subsection the time influence of generating L2 read misses is evaluated. The evaluation is presented using a figure and backed up with a small validation of the statistical distribution of the measurement results.

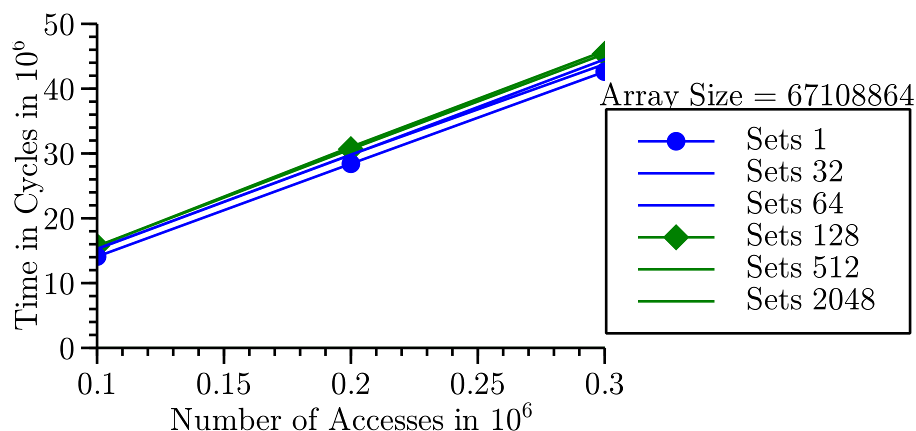


Figure 9.18: Time Behavior of L2 Read Cache Miss Events

In Figure 9.18 the time behavior is presented. Here, an array size of 67108864 bytes has been used and the *sets* parameter was varied from 1 to 2048 by doubling its value.

In the figure, it can be seen that the time increases linearly with number of accesses. This behavior is deterministically for the different *sets* values. Again, only an excerpt of the results are presented for convenience.

Accesses	Time Average	SCV
100000	154.47	0.00191
200000	152.76	0.00189
300000	151.58	0.00188

Table 9.15: Time Evaluation for L2 Cache Read Miss Events

In Table 9.15 the number of accesses is presented in the first column. In the second column the average time for one access is shown and the SCV for calculating the average value is depicted in the third column.

As can be seen, the time spent for producing a L2 read miss, which is a main memory access, is approximately 152 cycles with small variations only. Hence, the time behavior is deterministically for the different number of accesses. Additionally, the results are consistent with the access time of the main memory in the system. Here, an access takes around 150 cycles. The access times of the main memory have been studied and are presented in Figure 3.1).

Conclusion To summarize, this section presents that a specified amount of cache events can be created in a predefined cache level by using the *dcm PSF* as described in Chapter 7.

The evaluation has been done twofold. First, a time evaluation has been done. Second, oprofile measurements were performed if possible. Additionally, the results of the time measurements were compared to the systems parameters.

It has been shown that a specified amount of cache events can be created. This can be done for read as well as for write events. Moreover, it is possible to create misses and hits as predefined. The results show a deterministic and reproducible behavior. Because of the linear characteristic of the graphs the parameters for the simulation can easily be evaluated and adjusted to the needs of the *DPS* study using *dcm stubs*.

9.5 Simulated Software Functionality

For validating the *libSSF*, the test environment as described in this chapter is used. The test cases were defined according to the requirements as specified in Sections 8.1 and 8.3.3.

9.5.1 Structure of the Test Application

The test application includes a deterministic algorithm to modify the members of an object by using the interface of the object class. The object, which is stubbed is presented in Figure 9.19.

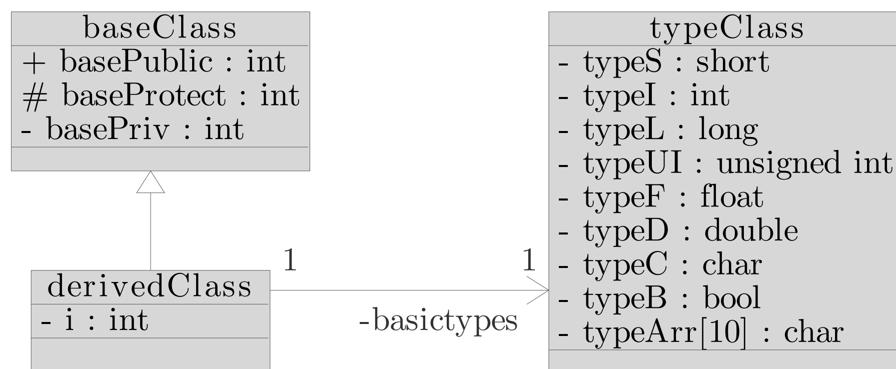


Figure 9.19: Structure of the Simulated Object

The type of the object, which is modified within the test case, is called “class derivedClass” and includes a private integer variable itself. Moreover, this object inherits the members of the “class baseClass”. These three members are integers with a visibility of the types “public”, “protected” and “private”. Additionally, the “derivedClass” object instances a variable of the type “class typeClass”. This includes members for basic data types. All of these members are declared as private.

Using this structure for the test object has several advantages. It can be determined whether the members of different visibility types can be stubbed. Moreover, it can be validated whether the basic data types can be used as well as a arrays. Additionally, the inheritance of objects is included in the test case. Last but not least, it can be evaluated whether objects can be recursively created within the *libSSF* in the case if a constructor call is stubbed (see Section 8.3.3).

9.5.2 Test Cases and Test Case Evaluation

The first step of the test case evaluation is to execute the original algorithm and to trace the values of the members of the objects. In the second step, the original algorithm is executed and the values are stored into a log file using the “record” functionality of the *libSSF*. This log file will then be examined and the results are compared with the results of the first step. As last step, the original algorithm is removed from the test execution and the values of the objects will be restored from the trace file using the *libSSF*. As before, the values of the objects are examined and compared to the the original test case execution.

These evaluations are done for the four different restore scenarios as described in Section 8.3.3:

1. Object is initialized as well as data are stored in the trace file. This behavior is the most common case as the CUS is usually changing the values of the objects.

2. The object is not initialized and no data are stored in the trace file. This case can also be common, e.g., if a search function does not find the entry and only returns a “NULL” reference.

3. The object is not initialized but data are stored in the trace file. This behavior can be interpreted as the constructor of the object is called within the CUS.

4. An initialized object is available but no data are stored in the trace file. This behavior cannot be handled in the *libSSF*. Here, the execution of the application is terminated.

The above described test case scenarios as well as the test case structure covers a broad variety of use cases for the application of the *libSSF*. Especially, the different visibilities of members of the object are evaluated. Additionally, the different basic types as well as inheritance of classes is included.

Conclusion The evaluation has shown that the *libSSF* is capable to simulate the functional behavior of an algorithm for a deterministic test case scenario. Here, the CUS can be replaced by a stub, which restores the previously recorded behavior of the software functionality. All test and use case scenarios as described in this section has been passed.

Chapter 10

Conclusion and Future Work

This chapter summarizes the findings of the presented approach. It highlights the contribution and critically reviews the approach of the dynamic performance stubs' framework. Additionally, future work is established, which also addresses some of the critical review items.

10.1 Conclusion

This thesis presents a novel approach for a software performance engineering framework and methodology, called *dynamic performance stubs*. The main contribution is that they enhance the methods of performance tuning and measurement by providing a cost-benefit analysis of different optimization approaches for a software bottleneck within a particular system. Hence, the final optimization of the bottleneck can be realized well-founded and gain-oriented. This can be achieved by replacing the software bottleneck with a *dynamic performance stub* that simulates the functional as well as the performance behavior of the bottleneck. Here, the performance behavior of the stub can be adjusted to study several different optimization levels. Such, the optimization's influence on the overall system can be measured before doing the optimization.

In addition to the cost-benefit analysis of an optimization, many further advantages (see Section 4.8) and extended evaluations (see Section 4.7) can be achieved by using *dynamic performance stubs*. These are summarized in the following non-exhaustive list:

- **Advantages:**

  "Hidden" bottleneck detection: This can be achieved as the bottleneck can be replaced by a *particular dynamic performance stub*, which is adjusted to consume almost no system's resources. So, the "hidden" bottleneck becomes the new bottleneck and can be identified easily.

  Balance between optimization effort and achievable gain: As a cost-benefit analysis can be achieved, it is possible to optimize the bottleneck with a balance between effort and gain.

- **Extended evaluation:**

  Idealized measurements: In this case, the *particular dynamic performance stub* is adjusted to use almost no system's resources. Hence, a bottleneck, which has been optimized in an ideal way can be simulated.

  Load and stress tests: This is the opposite to the "idealized measurements". Here, the *particular dynamic performance stub* is adjusted to strongly consume the system's resource. Hence, the load in the system can be significantly increased. This configuration can also be used to simulate stress tests.

System bounds evaluation: Here, an answer to the questions: "When does a system performance bounds changeover happen?" and "How much capabilities are remaining in this system's resource?" can be achieved. This can be done by varying the performance behavior of the *particular dynamic performance stub* at the system's performance bound. E.g., a changeover from a CPU bound system to a memory bound system can be realized by significantly increasing the memory utilization in the system.

Early optimization results: The gain of a possible optimization can be determined in advance. This determination of the gain leads to early optimization results.

These aspects can also be used to avoid the drawbacks of an over- or under-optimization. More advantages and extended evaluation methods can be found in Sections 4.7 and 4.8. Additionally, the items of the list above are described in more detail in these sections.

Contribution

In the following, the framework and the contributions of the *dynamic performance stubs*' approach are described. *Dynamic performance stubs* are used to replace a software bottleneck by a performance stub. The performance behavior of this stub can be dynamically adjusted to the needs of the performance improvement study; and thus, different performance optimization levels of the bottleneck can be simulated in the system. Here, the performance measurements, which identified the software bottleneck, are repeated to study the performance behavior of the system by using a *dynamic performance stub*.

The main contributions within the *dynamic performance stubs* (Chapter 4) are:

- Definition of the framework.
- Definition of a general methodology to apply the *dynamic performance stubs*.
- Definition of extended use cases to support the performance measurement and performance tuning process.

In order to replace the bottleneck, the *dynamic performance stub* has to simulate the functional behavior of the bottleneck for the predefined performance test cases. Moreover, the performance behavior has to be adjustable.

The framework of the *dynamic performance stubs* is subdivided into the *performance simulation functions*, which can be used to adjust the performance behavior regarding a dedicated system resource; and, the *simulated software functionality*. Those two elements are combined to built a *particular dynamic performance stub*, which simulates the performance behavior of a dedicated system resource. The following subsections list the contributions in the particular areas.

Performance Simulation Functions

The *performance simulation functions* are defined to simulate the performance behavior of the bottleneck regarding a separated aspect of the system resources. A classification of the different *performance simulation functions* can be found in Section 4.4. Within this thesis the possibility to simulate the performance behaviors for the system resources "CPU" (see Chapter 5), "main memory" (see Chapter 6) and "data cache memory" (see Chapter 7) have been evaluated.

CPU Performance Simulation Functions The *CPU performance simulation function* are used to simulate the performance behavior regarding the time consumption of a software algorithm. So, they can be used to simulate the CPU behavior of a process. The simulation is able to consume a dedicated amount of CPU time as well as to delay the execution of the process. Those two elements can be used to utilize the CPU with a constant load or to adjust the CPU utilization to a predefined value. Moreover, the *CPU performance simulation functions* can be used to simulate small or large time slices.

Main Memory Performance Simulation Functions By using *main memory performance simulation functions*, the allocation behavior of process' main memory can be simulated. So, it is possible to allocate and free different amounts of stack and heap memory at a given time during the simulation. Moreover, the memory is allocated in chunks, i.e., the process can extend or decrease its memory consumption without the necessity of freeing the memory space and allocating the new amount. Additionally, it is also possible to change only one aspect, e.g., heap memory, of the main memory allocation. Finally, the main memory is used create the necessary amounts of page faults are created, too.

Data Cache Memory Performance Simulation Functions The *data cache memory performance simulation function* can be used to simulate the data

cache access behavior. Especially, they are able to simulate read and write accesses. These accesses can be realized as hits and misses on the specified cache level. Moreover, the *data cache memory performance simulation functions* can be adjusted to create an amount of cache accesses.

To support and improve the simulation results, *calibration functions* have been developed (see Sections 5.3, 6.3 and 7.3). These *calibration functions* adjusting the *performance simulation functions* to the target. Usually, the *calibration functions* have to be executed once for each system. The results can be stored and reused if further performance evaluations using *dynamic performance stubs* are necessary.

Simulated Software Functionality

To simulate the functional behavior of the bottleneck, the *simulated software functionality* (see Chapter 8) has been introduced. The methodology is split into two main steps: "record" and "restore" the functional behavior of the bottleneck.

In a first step, the functional behavior of the bottleneck for the predefined performance test cases configuration is recorded. This recording is realized by storing the input and output values of the software functionality or algorithm.

As the functional behavior of the bottleneck has to be deterministic for the same test cases, the recorded behavior can be used to simulate the algorithm. Hence, the stored results are restored into the application memory in a second step.

Finally, the application as well as the test environment can be used to validate the functional behavior of the *simulated software functionality.*

The approach is described in the methodology subsection. A possible realization of the *simulated software functionality* is presented in Section 8.3. Here, a library, called *libSSF*, shows the application of the *simulated software functionality* to the software. A case study as well as a discussion concludes the approach of the *simulated software functionality.*

Particular Dynamic Performance Stubs

By combining the *performance simulation functions* with the *simulated software functionality*, *dynamic performance stubs* can be built to simulate a dedicated system resource, e.g., CPU. This combination leads to a *particular dynamic performance stub.*

Within each of the "stubs" chapters (Chapters 5, 6 and 7), the requirements for the *performance simulation functions* and a methodology are provided. Moreover,

the applications of the *particular dynamic performance stubs* are presented by a case study.

It has been shown that *dynamic performance stubs* can be used to evaluate the systems performance behavior of different performance optimization approaches. Moreover, by dynamically adjusting the performance parameters a cost-benefit analysis can be realized. Thus, the performance optimization of a bottleneck can be done based on well-founded measurements instead of pure estimations.

Critical Remarks

We are aware that it is not always possible to realize the optimization with the optimal value. So, over- and under-optimizations can still take place. But, this does not lower our approach as a well-founded cost-benefit analysis as well as accurate estimations of the possible gain always support and improve the performance optimization process.

Maybe the effort to apply our methodology outweighs the amount of effort needed for the realization of the software bottleneck itself. However, by using *dynamic performance stubs* many additional advantages, e.g., deeper system knowledge, can be achieved. So, the results of the *dynamic performance stubs* study can additionally be used to determine further optimization potentials. Moreover, “hidden” bottlenecks can be detected, which provides a clear indication for further optimizations potentials.

A performance optimization study with the approach of the *dynamic performance stubs* is usually done late in the software development cycle as it is based on the available application as well as the test environment. A result could be that it is not possible to optimize the performance bottleneck to the level requested by the performance target specification; i.e., a necessary redesign of the software is indicated. Beside of being an advantage of the *dynamic performance stubs'* approach, this information should have been available much earlier in the development process.

As for most performance tuning approaches, the results of an inaccurate *dynamic performance stubs* optimization study can be misleading and wrong conclusions can be drawn. This is particularly important if the results indicate that a software redesign is necessary to achieve the requirements of the performance target specification. Here, much unnecessary effort can be spent.

Our realizations of the presented *performance simulation functions* and *simulated software functionality* highly depend on a particular system as well as on the

programming language C++. However, the realization should be easily adjustable to further system architectures as well as programming languages. Moreover, our approaches and methodologies can easily be extended to various different systems.

10.2 Future Work

The *dynamic performance stubs* approach affects many different fields of work. Due to that the amount of work cannot be fully achieved in the scope of a single PhD-Thesis. Even in the areas addressed within this thesis several different extensions and improvements can be realized.

This section is split into three main areas. First, future work in areas, being addressed in this thesis is presented. The second part lists future research directions for the *performance simulation functions*. Finally, future work to extend the *dynamic performance stubs* approach to further types of software systems is discussed.

Extensions to the Discussed Areas

In this section, future work targeting to improve the *dynamic performance stubs* framework and methodology as discussed in this thesis is evaluated.

Data Cache Memory Performance Simulation Functions

Our described approach to simulate the data cache memory access behavior is based on the assumption that constantly accessing different memory locations of an array will cause cache hits or misses within the desired cache level.

As the cache prefetching algorithms in modern CPU are partly capable to predetermine the next data elements, which will be accessed, the introduced algorithm needs large arrays to be able to work sufficiently. This, of course, leads to an unnecessary high memory utilization in the system.

Moreover, the “out-of-order execution” of modern CPUs is able to fetch more than one datum at the same time. This results in more cache hits as expected by the pure analysis of the *data cache performance simulation functions*.

An investigation towards an improved *data cache memory performance simulation functions* is considered as future work.

Simulated Software Functionality

The *simulated software functionality* is realized by a prototype called *libSSF*. This library is based on a proof of concept to validate the main functional behavior of the approach. There are two major improvement areas.

First, the library mainly allows to store and restore the values of the C++ basic data structures, e.g., basic data types, arrays, structs and classes. Hence, the library can be heavily extended by providing a solution to stub more sophisticated data structures. Especially, the types “list”, “vector” or “stack” from the standard template library (STL) should be supported.

Second, the influences of the *libSSF* to the time and memory behavior have to be evaluated and improved. By now, the library loads all data, which will be restored, in the start up phase. This behavior should be adjustable. So, the number of preloaded data entries can be specified. All other data will be loaded in chunks on demand.

The library *libSSF* used within this thesis targets more at a proof of concept than a well-designed software component. Hence, a redesign of the library was started at the end of the writing period of this thesis. Further investigations have shown that the older version (which is presented in this thesis) is not capable to correctly store and restore classes derived from virtual classes. The reason is that these classes have a hidden member called virtual table pointer (vpointer), which is a pointer to the virtual method table (vtable), to support the dynamic dispatch method. Our investigations have shown that it is also possible to stub these classes. But, this is only supported in our new version of the *libSSF*; but not, in the version described in this thesis.

Extensions of the Methodology

The methodologies of the *main memory* - and *data cache memory stubs* (Sections 6.5 and 7.4) mainly provide the basic steps. These can be extended and have to be validated in industrial case studies. Additionally, some steps to support the performance analyst on changing the performance behavior of the stub best have to be evaluated.

Further Performance Simulation Functions

In Figure 4.3 the different *performance simulation functions* are depicted. The light gray sections are considered as future work. The following list presents the unresolved types of *performance simulation functions*:

- Memory performance simulation functions
  - Instruction cache
  - TLB cache
- I/O performance simulation functions
- Network performance simulation functions

In the following paragraphs, some first suggestions in the area of *memory performance simulation functions* are provided. An overview of the performance behaviors, which will be simulated in the remaining *performance simulation functions* can be found in Section 4.4.

Instruction Cache Memory Performance Simulation Functions

The *instruction cache memory stubs* will simulate the amount of instruction cache miss events. Hence, an algorithm, which prevents the instruction cache prefetcher from loading the next instruction has to be evaluated. Instruction cache misses can be achieved by "self-modifying code" (SMC) (see [10]). In this case, the code that will be executed next is constantly overwritten by the *performance simulation functions*. Hence, the prefetcher is not able to load the next instruction in advance. More information about instruction prefetchers can be found in [34, 110].

TLB Cache Memory Performance Simulation Functions

In [79] a similar approach as the newly introduced *data cache memory performance simulation functions* is used to determine the size of the TLB cache. In this case, the time behavior to load the data is studied to evaluate whether a TLB hit or miss occurred in the system. Hence, the approach creates TLB hits and misses. So, the presented *data cache memory performance simulation functions* (Section 7.2) can be adjusted to create TLB hits and misses by changing the accessstride value. A more sophisticated analysis as well as a thorough evaluation of the *TLB cache memory performance functions* is considered as future work.

Extension to Different Architectures

The approaches of the *dynamic performance stubs* framework and methodology will be extended to provide the stubs to a broader field of application.

Our *CPU performance simulation functions* are already in line with multi-core systems; but, still suppose the bottleneck to be single-threaded. Moreover, the optimization of the bottleneck is supposed to be single-threaded, too. An extension to support the software performance engineer with evaluations for a multi-threaded optimization of the bottleneck should be evaluated.

Moreover, the *dynamic performance stubs* will be extended to embedded systems. The main difference between our approach and the application of *dynamic performance stubs* in embedded systems is basically the lack of sufficient performance measurement tools. Here, the methodologies and the *simulated software functionality* have to be adjusted to meet the requirements of a embedded systems.

Another category of systems in which *dynamic performance stubs* will be used in future are virtual systems, e.g., systems running in a virtual machine (VM). Here, several questions arise. Most important to mention is that many performance analysis tools measure obscure results as the kernel of the system does not have direct influence on the hardware. More closer, the provided amount of system resources, e.g., CPU cycles, for the system under study is adjusted by the VM. This leads to a non-deterministic performance behavior within the virtual system.

Finally, our approach of creating and using *dynamic performance stubs* is based on the simulation of the performance behavior of a single software resource. More investigations in composing different *particular dynamic performance stubs* have to be done. Additionally, an aggregation, which interleaves different *performance simulation functions* within a single stub, can be evaluated.

Tools Overview

Callgrind	Valgrind Tool `valgrind.org/info/tools.html`
Dtrace	OpenSolaris Community - DTrace `opensolaris.org/os/community/dtrace`
EEL	Executable Editing Library `pages.cs.wisc.edu/~larus/eel.html`
GNU gprof	The GNU Profiler `gnu.org/software/binutils/manual/gprof-2.9.1/htmlmono/gprof.html`
LKST	Linux Kernel State Tracer `lkst.sourceforge.net`
LTT	Linux Trace Toolkit `www.opersys.com/LTT`
LTTng	Linux Trace Toolkit Next Generation `ltt.polymtl.ca`
Massif	Massif - A Heap Profiler `valgrind.org/info/tools.html`
OProfile	OProfile - A System Profiler for Linux `oprofile.sourceforge.net`
paradyn	Paradyn Parallel Performance Tools `www.paradyn.org`
Perfctr	Perfctr - Linux Performance Counters Driver `perfctr.sourceforge.net`
PerfSuite	PerfSuite - Collection of Performance Analysis Software `perfsuite.sourceforge.net`
PIN	Pin - A Binary Instrumentation Tool `www.pintool.org`
Rational Quantify	IBM Rational Quantify

	`www.ibm.com/software/awdtools/quantify/support`
SPEC CPU2006	Standard Performance Evaluation Corporation
	`www.spec.org/cpu2006`
Systemtap	Systemtap
	`sourceware.org/systemtap/`
TAU	Tuning and Analysis Utilities
	`www.cs.uoregon.edu/research/tau`
Valgrind	Instrumentation Framework - Valgrind
	`valgrind.org`

Abbreviations

CF	Calibration Functions
CSR	Cache Set References
CUS	Component Under Study
DPS	Dynamic Performance Stubs
ICSR	Identical Cache Set References
ISD	Intermediate Simulation Data
MD	Measured Data
PDPS	Particular Dynamic Performance Stubs
PSF	Performance Simulation Functions
RTC	Real Time Clock
SCV	Squared Coefficient of Variation
SD	Simulation Data
SUT	System Under Test
SPE	Software Performance Engineering
SSF	Simulated Software Functionality
TSC	Time Stamp Counter

Bibliography

[1] F. Agner. 1. Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms. online: `http://www.agner.org/optimize`, 2006. [last update: July 5, 2006].

[2] F. Agner. 2. Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms. online: `http://www.agner.org/optimize`, 2006. [last update: July 5, 2006].

[3] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. ACM Press, 2004.

[4] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Delay Metric for IPPM. RFC 2679 (Proposed Standard), September 1999.

[5] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Packet Loss Metric for IPPM. RFC 2680 (Proposed Standard), September 1999.

[6] A. Bertolino and E. Marchetti. *Software Engineering: The Development Process - A Brief Essay on Software Testing*, volume 1, chapter 7, pages 393–411. John Wiley & Sons, Inc., third edition, 2005.

[7] K. Beyls and E. H. D'Hollander. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

[8] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly, third edition, 2005.

[9] T. Brekne, M. Clemetsen, P. Heegaard, T. Ingvaldsen, and B. Viken. State of the Art in Performance Monitoring and Measurements. Technical report, Telenor Communication, May 2002.

[10] H. Cai, Z. Shao, and A. Vaynberg. Certified Self-Modifying Code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, 2007.

[11] S. Chapman. Finding the Critical Path - A Simple Approach. In *CMG '09: International Conference Proceedings*. Computer Measurement Group, 2009.

[12] Pong P. Chu and Ramana Gottipati. Write Buffer Design for On-Chip Cache. In *Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, ICCS '94, pages 311–316, Washington, DC, USA, 1994. IEEE Computer Society.

[13] M.L. Collard, H.H. Kagdi, and J.I. Maletic. An xml-based lightweight c++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 134 – 143, May 2003.

[14] D. E. Comer and R. E. Droms. *Computer Networks and Internets with Internet Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2001.

[15] Intel Corporation. *A Detailed Look Inside the Intel®NetBurst™Micro-Architecture of the Intel Pentium®4 Processor*, November 2000. online: http://people.virginia.edu/~zl4j/CS854/netburst_detail.pdf.

[16] Intel Corporation. *IA-32 Intel®Architecture Software Developer's Manual Volume 3*, 2003.

[17] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manuals Volume 3A*, November 2008. online: http://developer.intel.com/products/processor/manuals/index.htm.

[18] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manuals Volume 3B*, November 2008. online: http://developer.intel.com/products/processor/manuals/index.htm.

[19] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[20] CPPX. CPPX - Open Source C++ Fact Extractor. online: `http://www.swag.uwaterloo.ca/cppx`. January 15, 2011.

[21] B. Dawes, D. Abrahams, and R. Rivera. Boost C++ Libraries. online: `http://www.boost.org/`. January 15, 2011.

[22] D. DeGroot, editor. *Computer Architecture News*, volume 35. ACM, March 2007.

[23] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393 (Proposed Standard), November 2002.

[24] R. Demming and D. J. Duffy. *Introduction to the Boost C++ Libraries*, volume I - Foundations. Datasim Education, 2010.

[25] P. A. Dinda. The Statistical Properties of Host Load. *IOS Press*, 7(3-4):211–229, 1999.

[26] P. A. Dinda and D. R. O'Hallaron. Host Load Prediction Using Linear Models. *Cluster Computing*, 3(4):265–280, 2000.

[27] T. B. D'Orazio. *Programming in C++: Lessons and Applications*. McGraw-Hill, Boston, MA, USA, international edition edition, 2004.

[28] U. Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat, Inc., November 2007.

[29] H. Dwyer and H. C. Torng. An Out-Of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 272–281, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[30] B. Eckel. *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2005.

[31] Y. Etsion and D. Feitelson. Time Stamp Counters Library - Measurements with Nano Seconds Resolution. Technical Report 2000-36, The Hebrew University of Jerusalem, 2000.

[32] P. Ezolt. A Study in Malloc: A Case of Excessive Minor Faults. In *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*, pages 17–17, Berkeley, CA, USA, 2001. USENIX Association.

[33] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[34] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Instruction Fetch Streaming. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 1–10, Washington, DC, USA, 2008. IEEE Computer Society.

[35] P. J. Fortier and H. E. Michel. *Computer Systems Performance Evaluation and Prediction*, volume 1. Digital Press, Burlington, 2003.

[36] A. Fugmann. Scheduling Algorithms for Linux. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2002.

[37] GCC-XML. GCC-XML, the XML output extension to GCC! online: `http://www.gccxml.org/`, 2011. [January 15, 2011].

[38] Optimize Options - Using the GNU Compiler Collection (GCC). online: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, 2007. [April 20, 2007].

[39] P. Genua. A Cache Primer. Technical report, Freescale Semiconductor, Inc., Austin, TX, USA, 2004. AN2663, Rev.1.

[40] George E. P. Box, Gwilym M. Jenkins and Gregory C.Reinsel. *Time series analysis: forecasting and control.* Prentice Hall, Englewood Cliff/N.J., third edition, 1994.

[41] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, second edition, 2003.

[42] N. H. Gunther. *The Practical Performance Analyst.* McGraw-Hill Education, 1998.

[43] F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 228–239, New York, NY, USA, 2006. ACM Press.

[44] J. Handy. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[45] P. Heidelberger and S. S. Lavenberg. Computer Performance Evaluation Methodology. *IEEE Transactions on Computers*, 33(12):1195–1220, 1984.

[46] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 12(38):1612–1630, 1989.

[47] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor, 2001.

[48] J. Hughes. Performance Engineering throughout the System Life Cycle. Technical report, SES Inc., 1998. [April 04, 2007].

[49] IEEE. IEEE Standard Glossary of Software Engineering Terminology: IEEE Std 610.12-1990. Technical report, IEEE, 1990.

[50] Intel Corporation. *IA-32 Intel® ArchitectureSoftware Developer's Manual - SystemProgramming Guide*, June 2003.

[51] Intel Corporation. *IA-32 Intel® ArchitectureSoftware Developer's Manual - SystemProgramming Guide, Part I*, June 2006.

[52] Intel Corporation. *Intel®64 and IA-32 Architectures Optimization Reference Manual*, 2006.

[53] J. Weidendorfer J. Seward, N. Nethercote and the Valgrind Development Team. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Limited, 2008.

[54] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufman Publ. Inc., 2007.

[55] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley and sons, Inc., 1991.

[56] A. Jayasumana, N. Piratla, T. Banka, A. Bare, and R. Whitner. Improved Packet Reordering Metrics. RFC 5236 (Informational), June 2008.

[57] E. B. John, S. Petko, L. K. John, and J. Law. Access Time and Energy Tradeofs for Caches in High Frequency Microprocessors. In *Circuits and Systems, 2002*, MWSCAS-2002, pages 421–424, 2002.

[58] L. K. John. Performance Evaluation: Techniques, Tools and Benchmarks. In *The Computer Engineering Handbook*, pages 8–20 – 8–36. CRC Press, 2002.

[59] T. John and R. Baumgartl. Worst Case Behavior of CPU Caches. In *6th Real-time Linux Workshop*, 2006.

[60] S. C. Johnson and D. M. Ritchie. The C Language Calling Sequence. Computing Sience Technical Report No. 102, September 1981.

[61] N. P. Jouppi. Improving Direct-mapped Cache performance by the Addition of a Small Fully-Associative Cache Prefetch Buffers. In *25 years of the International Symposia on Computer Architecture (selected papers)*, ISCA '98, pages 388–397, New York, NY, USA, 1998. ACM.

[62] B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* Addison-Wesley Professional, 2005.

[63] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall International, second edition, 1988.

[64] M. Kettner. *Fehlerdiagnose und Problembehebung unter Linux.* SUSE Press, second edition, 2004.

[65] F. Khan. *LTE for 4G Mobile Broadband: Air Interface Technologies and Performance.* Cambridge University Press, first edition, 2009.

[66] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wiley, Reading, Mass., third edition, 1997.

[67] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wiley, Reading, Mass., second edition, 1998.

[68] R. Koodli and R. Ravikanth. One-way Loss Pattern Sample Metrics. RFC 3357 (Informational), August 2002.

[69] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. A Tool Chain for Reverse Engineering C++ Applications. *Sci. Comput. Program.*, 69(1-3):3–13, 2007.

[70] A. R. Lebeck. *Tools and Techniques for Memory System Design and Analysis*. PhD thesis, University of Wisconsin - Madison, 1995.

[71] M. E. Lee. Optimization of Computer Programs in C. online:`http://www.prism.uvsq.fr/~cedb/local_copies/lee.html`, 1999. April 25, 2007.

[72] C. Lever and D. Boreham. malloc() Performance in a Multithreaded Linux Environment. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association.

[73] P. Liggesmeyer. *Software-Qualität : Testen, Analysieren undVerifizieren von Software*. Spektrum Akademischer Verlag GmbH, Berlin, 2002.

[74] D. J. Lilja. *Measuring Computer Performance: A Practioner's Guide*. Cambridge University Press, New York, 2000.

[75] R. Love. *Linux-Kernel-Handbuch*. Addison-Wesley Verlag, 2005.

[76] LTE Performance and Capacity Management Group of our Industrial Partner. LTE RL T Systems Performance SFS. Unpublished, 2008.

[77] LTE Performance and Capacity Management Group of our Industrial Partner. LTE Traffic Model (Specification). Unpublished, 2008.

[78] Linux Trace Toolkit Next Generation. online: `http://lttng.org`, 2010. [March 15, 2010].

[79] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.

[80] S. Manegold and P. Boncz. Cache-Memory and TLB Calibration Tool. online: `http://homepages.cwi.nl/~manegold/Calibrator`, 2009. [February 11, 2009].

[81] J. J. Marciniak. *Encyclopedia of Software Engineering*. John Wiley & Sons Inc, second edition, 2002.

[82] G. Marin. *Application Insight Through Performance Modeling.* PhD thesis, Rice University, 2007.

[83] G. Marin and J. Mellor-Crummey. Application Insight Through Performance Modeling. In *26th IEEE International Performance Computing and Communications Conference (IPCCC'07)*, New Orleans, April 2007.

[84] S. McConnell. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, Redmond, WA, USA, second edition, 2004.

[85] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. Packet Reordering Metrics. RFC 4737 (Proposed Standard), November 2006.

[86] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, 2004.

[87] J. M. Newcomer. Optimization: Your Worst Enemy. online: `http://www.codeproject.com/tips/optimizationenemy.asp?print=true`, August 2000. [May 02, 2007].

[88] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330 (Informational), May 1998.

[89] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee. Reference Distance as a Metric for Data Locality. In *HPC-ASIA '97: Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, Washington, DC, USA, 1997. IEEE Computer Society.

[90] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. *Software Maintenance, IEEE International Conference on*, 0:188–197, 2004.

[91] R. Ramey. Serialization - Overview. online: `http://www.boost.org/libs/serialization`. January 15, 2011.

[92] D. J. Reifer. The Smart Stub as a Software Management Tool. *SIGSOFT Softw. Eng. Notes*, 1(2), 1976.

[93] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Predictability of Cache Replacement Policies. Technical report, Universität des Saarlandes, Saarbrücken, September 2006.

[94] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 38–49, New York, NY, USA, 1998. ACM.

[95] J. B. Rothman and A. J. Smith. Sector Cache Design and Performance. Technical Report UCB/CSD-99-1034, EECS Department, University of California, Berkeley, Jan 1999.

[96] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. Technical Report 172, University of York, 1992.

[97] A. Schmietendorf and A. Scholz. *Aspects of Performance Engineering - An Overview*, pages IX – XII. Springer Verlag Berlin, Heidelberg, 2001.

[98] C. B. Sears. The Elements of Cache Programming Style. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, Berkeley, CA, USA, 2000. USENIX Association.

[99] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley, seventh edition, 2005.

[100] S. S. Skiena. *The Algorithm Design Manual*. Springer, Berlin, November 1997.

[101] C. U. Smith. Increasing Information Systems Productivity by Software Performance Engineering. In *Int. CMG Conference*, 1981.

[102] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[103] C. U. Smith. Software Performance Engineering. In *Encyclopedia of Software Engineering*, pages 1545–1562. John Wiley & Sons Inc, 2002.

[104] C. U. Smith. Formal Methods for Performance Evaluation. In *Introduction to Software Performance Engineering: Origins and Outstanding Problems*, pages 395 – 428, 2007.

[105] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[106] C. U. Smith and L. G. Williams. Best Practices for Software Performance Engineering. In *Int. CMG Conference*, pages 83–92, 2003.

[107] S. Sodhi and J. Subhlok. Automatic Construction and Evaluation of Performance Skeletons. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 88.1, Washington, DC, USA, 2005. IEEE Computer Society.

[108] S. Sodhi, J. Subhlok, and Q. Xu. Performance Prediction with Skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[109] I. Sommerville. *Software Engineering*. Pearson Studium, sixth edition, 2001.

[110] L. Spracklen, Y. Chou, and S. G. Abraham. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 225–236, Washington, DC, USA, 2005. IEEE Computer Society.

[111] R. Srinivasan and O. Lubeck. MonteSim: A Monte Carlo Performance Model for In-order Microarchitectures. *ACM SIGARCH Computer Architectur News*, 33(5):75–80, December 2005.

[112] E. Stephan. IP Performance Metrics (IPPM) Metrics Registry. RFC 4148 (Best Current Practice), August 2005.

[113] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Longman, Amsterdam, second (rev) edition, 2008.

[114] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[115] J. Subhlok and Q. Xu. Automatic Construction of Coordinated Performance Skeletons. In *IPDPS*, pages 1–5, 2008.

[116] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., second edition, 2001.

[117] W. Tansey. Automated Adaptive Software Maintenance: A Methodology and Its Applications. Master's thesis, Virginia Tech, May 2008.

[118] W. Tansey and E. Tilevich. Efficient Automated Marshaling of C++ Data Structures for MPI Applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, apr. 2008.

[119] A. Toomula and J. Subhlok. Replicating Memory Behavior for Performance Prediction. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–8, New York, NY, USA, 2004. ACM.

[120] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 469–480, Washington, DC, USA, 2005. IEEE Computer Society.

[121] P. Trapp and C. Facchi. Performance Improvement Using Dynamic Performance Stubs. Technical Report 14, Fachhochschule Ingolstadt, August 2007.

[122] P. Trapp and C. Facchi. How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs. In *CMG '08: International Conference Proceedings*, pages 343 – 353, 2008.

[123] P. Trapp and C. Facchi. Main Memory Stubs to Simulate Heap and Stack Memory Behavior. In *CMG '10: International Conference Proceedings*. Computer Measurement Group, 2010.

[124] P. Trapp, C. Facchi, and S. Bittl. The Concept of Memory Stubs as a Specialization of Dynamic Performance Stubs to Simulate Memory Access Behavior. In *CMG '09: International Conference Proceedings*. Computer Measurement Group, 2009.

[125] P. Trapp, C. Facchi, and M. Meyer. Echtzeitverhalten durch die Verwendung von CPU Stubs: Eine Erweiterung von Dynamic Performance Stubs. In *Workshop "Echtzeit 2009 - Software-intensive verteilte Echtzeitsysteme", Informatik Aktuell*, pages 119–128. Springer Verlag, 2009.

[126] P. Trapp, M. Meyer, and C. Facchi. Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs. In *ICSEA '10: Proceedings of the International Conference on Software Engineering Advances*. IEEE Computer Society, 2010.

[127] P. Trapp, M. Meyer, and C. Facchi. Dynamic Performance Stubs to Simulate the Main Memory Behavior of Applications. In *SPECTS '11: Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. IEEE Communications Society, 2011.

[128] P. Trapp, M. Meyer, and C. Facchi. How to Correctly Simulate Memory Allocation Behavior of Applications by Calibrating Main Memory Stubs. Technical Report 20, Hochschule Ingolstadt, May 2011.

[129] P. Trapp, M. Meyer, C. Facchi, H. Janicke, and F. Siewe. Building CPU Stubs to Optimize CPU Bound Systems: An Application of Dynamic Performance Stubs. *International Journal on Advances in Software*, 4(1&2), 2011. (accepted for publication).

[130] D. Tsafrir. The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops). In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 4, New York, NY, USA, 2007. ACM Press.

[131] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[132] R. van der Pas. Memory Hierarchy in Cache-Based Systems. Technical report, Sun Microsystems, Inc., November 2002.

[133] S. Venkataramaiah and J. Subhlok. Performance Prediction for Simple CPU and Network Sharing. In *LACSI Symposium 2002*, 2002.

[134] L. G. Williams and C. U. Smith. Five Steps to Solving Software Performance Problems. Technical report, Software Engineering Research and Performance Engineering Services, 2002.

[135] J. Wolf. *C von A bis Z. Das umfassende Handbuch.* Galileo Press, third edition, 2009.

[136] P.-C. Wu and K.-C. Huang. Case Studies on Cache Performance and Optimization of Programs with Unit. In *Software - Practise and Experience*, pages 167–172. John Wiley & Sons, Ltd., 1997.

[137] Q. Xu and J. Subhlok. Construction and Evaluation of Coordinated Performance Skeletons. Technical Report UH-CS-08-09, University of Houston, 2008.

[138] M. Zahran. Cache Replacement Policy Revisited. In *The Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) held in conjunction with the International Symposium on Computer Architecture (ISCA)*, June 2007.

[139] M. M. Zahran, K. Albayraktaroglu, and M. Franklin. Non-Inclusion Property in Multi-Level Caches Revisited. *I. J. Comput. Appl.*, 14(2):99–108, 2007.

[140] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor Hardware Counter Statistics as a First-Class System Resource. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[141] Y. Zheng, B. T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–96, Washington, DC, USA, 2004. IEEE Computer Society.