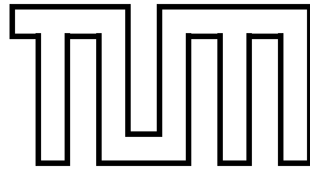


FAKULTÄT FÜR INFORMATIK

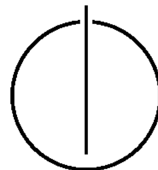
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

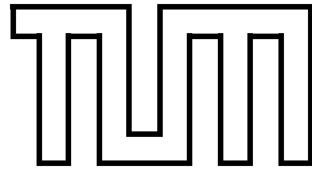
Dissertation

Automatic Characterization of Performance Dynamics with Periscope

Yury Oleynik

Technische Universität München





FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Automatic Characterization of Performance Dynamics with Periscope

Yury Oleynik

Vollständiger Abdruck der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Michael Bader

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Hans Michael Gerndt
2. Univ.-Prof. Dr. Wolfgang E. Nagel, Technische Universität Dresden

Die Dissertation wurde am 23.06.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.02.2015 angenommen.

Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17.06.2014 Yury Oleynik

Abstract

It was observed that for some High Performance Computing (HPC) applications the location and severity of performance bottlenecks change over the course of execution. With applications and hardware becoming more complex and, in particular, more dynamic, such effects will become even more widespread and paramount. Understanding of temporal performance dynamics becomes, therefore, important in the process of the development and tuning of HPC applications.

However, extending performance measurements into the temporal dimension leads to linearly growing overheads, size of collected data, visualization volumes, and analysis efforts. This makes manual investigation tedious and motivates the development of tools providing valuable insights into the dynamic characteristics of performance and at the same time mitigating the challenges mentioned above.

This thesis presents concepts and an iplementation allowing automatic analysis of temporal performance dynamics. For the first time in the field, advanced signal processing algorithms are applied in order to evaluate dynamic properties of performance bottlenecks. The approach significantly reduces the analysis efforts by the user by detailing the location, severity and high-level qualitative description of the relevant performance degradation trends. Alternatively, the results can be used to shrink the visualization volume by limiting it to the relevant time intervals.

Furthermore, the thesis presents a novel scheme for collection and analysis of dynamic profiles. The technique extends dynamic phase profiling by an on-line retrieval and on-line remote analysis of measurements. When compared to other approaches, such as compression and post-mortem analysis or visualization, the approach allows to solve the problem of the linearly growing size of stored data by making it independent from the time dimension.

In order to mitigate the overheads introduced by the direct source-level instrumentation, a novel dynamic instrumentation adaptation mechanism is presented. It is based on instrumentation strategies that automatically evaluate and adapt the granularity of the instrumentation based on the severity of the introduced overheads as well as the requirements of the iterative on-line analysis algorithm.

Acknowledgments

First and foremost, I would like to thank my supervisor Prof. Dr. Michael Gerndt for creating such a supportive and enthusiastic research environment and giving me an opportunity to realize my potential. I express my deepest gratitude for his wise and insightful guidance, for his unwavering support, for the trust and for the freedom he gave me in my work.

I would also like to thank my second supervisor, Prof. Dr. Wolfgang E. Nagel, for his valuable feedback and for a warm welcome at Technische Universität Dresden (TUD).

Furthermore, I would like to thank Prof. Dr. Arndt Bode, the leader of the Chair of Computer Architecture (LRR), for creating a friendly and creative atmosphere at the chair.

I would like to express my gratefulness to all my friends and colleagues at LRR. It was a pleasure and honor to be part of this community. Thank you for fruitful discussions, support and simply great time.

Furthermore, I would like thank all my former teachers and supervisors, notably Prof. Dr. Vladimir B. Parashin and Prof. Dr. Sergey I. Shchukin, who helped me build solid background in many fields which greatly served me in this work.

Special thanks go to my friends, Dr. Sergey Matushkin, Dr. Alexey Maystrou, Victor and Anastasia Rupp, for their support and great moments together.

Last but not least, I owe my deepest gratitude to my family for their love and care. You were the source of my strengths and determination in pursuing my goals through all the obstacles on the way. Thank you so much.

Yury Oleynik
Munich, Germany
2014

Contents

1 Introduction **1**

1.1 Contribution of This Work . . . 3

1.2 Study context: Periscope . . . 4

1.2.1 Analysis Automation . . . 4

1.2.2 Online Analysis . . . 4

1.2.3 Distributed Analysis . . . 5

1.2.4 Current Limitations . . . 6

2 Tools-Aided Performance Engineering **7**

2.1 Software Development Life-Cycle . . . 7

2.2 Performance Tuning Cycle . . . 7

2.3 Monitoring Performance . . . 9

2.3.1 Sampling . . . 10

2.3.2 Direct Instrumentation . . . 10

2.3.3 Measurement Overhead . . . 11

2.3.4 Mitigation of Overheads . . . 12

2.4 Performance Measurement . . . 13

2.4.1 Profiling vs Tracing . . . 13

2.4.2 Post-mortem vs Online Analysis . . . 14

2.5 Performance Analysis Automation . . . 15

2.6 Tools for Performance Engineering . . . 15

2.6.1 Score-P . . . 16

2.6.2 Scalasca . . . 17

2.6.3 Vampir . . . 19

2.6.4 ParaDyn . . . 20
2.6.5 Pathway . . . 22

3 State of the Art 25
3.1 Compression . . . 25
3.1.1 Clustering of Dynamic Profiles . . . 25
3.1.2 Wavelet Compression of Load Balance Measurements . . . 26
3.1.3 Compressed Complete Call Graphs . . . 27
3.2 Analysis . . . 29
3.2.1 Detection and Application Structure Extraction . . . 29
3.2.2 Root Cause Analysis . . . 31
3.3 Summary . . . 33

4 Automatic Instrumentation Adaptation 35
4.1 Introduction . . . 35
4.2 Overhead Model . . . 36
4.3 Instrumentation Strategies . . . 39
4.3.1 Total Overhead Reduction Strategy . . . 39
4.3.2 Prolog Overhead Reduction Strategy . . . 41
4.3.3 Analysis Guided Overhead Reduction Strategy . . . 43
4.4 Results . . . 43
4.4.1 Nested Loop Example . . . 44
4.4.2 PEPC . . . 47
4.5 Summary . . . 48

5 Temporal Scalability of Performance Dynamics Analysis 51
5.1 Introduction . . . 51
5.2 Design Overview . . . 52
5.3 Dynamic Phase Profiling . . . 53
5.4 Online Access Interface . . . 54
5.5 Online Processing of Temporal Performance Data . . . 55
5.6 Improved Periscope Analysis Engine . . . 57

5.6.1 Requesting and Storing Temporal Performance Data 58
5.6.2 Accessing Temporal Performance Data 59
5.6.3 Backward Compatibility with Legacy Properties 60
5.6.4 Handling Missing Values 60
5.6.5 Online Analysis 61
5.7 Summary 63

6 Automatic Analysis of Performance Dynamics 65
6.1 Motivation 65
6.1.1 Example Signal 66
6.1.2 Design Goals 67
6.2 Wavelet Analysis 68
6.2.1 Discrete Wavelet Transform 69
6.2.2 Implementation in Periscope 70
6.3 Qualitative Representation of Trends 71
6.3.1 Geometrical Interpretation 73
6.3.2 Scale-Space Filtering 73
6.3.3 Scale-Space Image 74
6.3.4 Scale-Space Qualitative Representation 76
6.3.5 Qualitative Summarization 78
6.3.6 Implementation in Periscope 78
6.4 Performance Dynamics Analysis Strategy 80
6.4.1 Design 80
6.4.2 Analysis Algorithm 81
6.4.3 Adapted APART Property Specification Language 84
6.4.4 Performance Dynamics Properties 84
6.5 Summary 92

7 Evaluation 95
7.1 Experimental Setup 95
7.2 CX3D - Czochralsky Crystal Growth Simulation 96
7.2.1 Automatic Analysis of Performance Dynamics with Periscope . . . 96

7.3 SPEC MPI2007 129.tera_tf . . . 99
7.3.1 Visual Analysis of Raw Temporal Performance Data . . . 100
7.3.2 Automatic Analysis of Performance Dynamics with Periscope . . . 102
7.4 PEPC . . . 104
7.4.1 Visual Analysis of Raw Temporal Performance Data . . . 105
7.4.2 Automatic Analysis of Performance Dynamics with Periscope . . . 107
7.5 INDEED . . . 109
7.5.1 Experimental data . . . 109
7.5.2 Automatic Analysis of Performance Dynamics with Periscope . . . 111

8 Summary and Outlook **115**
8.1 Future Work . . . 118

List of Figures

1.1 A schematic illustration of the Periscope analysis model. 5

1.2 A simplified schematic illustration of the Periscope architecture. 6

2.1 Performance Tuning Cycle Methodology. 8

2.2 Score-P architecture[8] 16

2.3 CUBE GUI[7] 18

2.4 Vampir GUI[11] 19

2.5 ParaDyn Time Histogram Display[6] 21

2.6 ParaDyn GUI[6] 22

2.7 Pathway workflow editor 23

2.8 Pathway experiment browser 24

3.1 A schematic process of the incremental on-line clustering into a maximum of 4 cluster. (i) The first four call-path profiles are collected and categorized in two equivalence classes. (ii) The profile 5 is collected and categorized to the equivalence class of profile 1. (iii) Call-paths 3 and 4 are merged together as they are the closest ones according to the distance function. [59]. 26

3.2 CCG of an example function[37]. For the compressed version see Figure 3.3 28

3.3 Compressed version of the example function shown in Figure 3.2 [37]. . . . 29

3.4 Time line diagram showing simplified event sequence on three processes. Shaded rectangles indicate certain activities defined by the corresponding events. These are circles for region enter and exits and squares for send and receive events. The arrows indicate the direction of the messages. It can be observed that extended execution time of the “comp” activity on process A leads to direct waiting time in the process B. This waiting time plus an additional delay caused by the receive operation on process B is then responsible for the waiting time in the process C [19]. 31

4.1 Pseudo code of the monitoring functions and the overheads. 37
4.2 Break down of the total, reported and pure times of a region nest (labeled as R1 and R2) together with the generated prolog and request overheads. . 38
4.3 Control flow of the TOR Strategy. 40
4.4 Specification of the “Excessive Total Relative Overhead“ property 41
4.5 Specification of the “Excessive Prolog Overhead“ property 41
4.6 Control flow of the AGOR strategy. 42
4.7 Example code with three nested loops. 45

5.1 Online dynamic profile collection and analysis scheme design. 52
5.2 Online temporal performance data processing scheme. 56
5.3 Data Provider class diagram. 58
5.4 Performance Data Base class diagram. 59
5.5 Simplified online analysis process flow diagram for the MPI wait-states analysis strategy. 61

6.1 Example signal plotting severity values of the property “Hot Spot of the Application” for 128 iterations of the CX3D progress loop. The values are normalized to the $[0, 1]$ interval. 66
6.2 Scaleogram of the example signal. It shows the energies of the wavelet coefficients in percentages to their sum plotted against corresponding scale and temporal location. 70
6.3 Geometric primitives for the 7 basic qualitative descriptors of the qualitative representation language. A: concave increase, B: concave decrease, C: convex decrease, D: convex increase, E: linear increase, F: linear decrease, G: constant 73
6.4 Qualitative representation of the smoothed example signal. The original signal is plotted with cyan bars; the zeros of the second and first derivatives of the smoothed signal are shown with red dots; the qualitative representation by both alphabetical labels and geometric primitives is plotted above with blue color. 74
6.5 Slices of the example signal’s Scale-Space Image at the given set of scales, σ. 75
6.6 Contours of zero-crossings of the example signal’s SSI. 76
6.7 Interval tree of the example signal’s SSI. 77
6.8 Class diagram of the SSF algorithm and related data-structures. 79
6.9 Algorithm of the Performance Dynamics Analysis Strategy 82

6.10 Specification of the “Significant Variability“ property 86
6.11 Grouping specification of two “Significant Variability“ properties 86
6.12 Specification of the ”Degradation Peaks” property 87
6.13 Grouping specification of two ”Degradation Peaks” properties 88
6.14 Specification of the ”Degradation Trends” property 90
6.15 Grouping specification of two ”Degradation Trends” properties 91
6.16 (a) Specification of the ”Degradation Pattern” property, (b) re-evaluation of the ”Degradation Pattern” property for the extended dynamic context . 93

7.1 Value map of the phase region execution time over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis). 101
7.2 Value map of the communication time in the `MPI_Wait` call over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis). . . . 102
7.3 Value map of the communication time in the `MPI_Allreduce` call over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis). 103
7.4 Value map of the communication time in `MPI_Allgather` call over the first 2048 iterations of the progress loop (x-axis) and 64 MPI ranks (y-axis). . . 105
7.5 Value map of the communication time in the `MPI_Alltoall` call over the first 2048 iterations of the progress loop (x-axis) and 64 MPI ranks (y-axis).106
7.6 CPU time of the time step iterations 110
7.7 Time step widths over the iterations of the time step loop 111

List of Tables

4.1 Phase region execution time during overhead estimation and the subsequent analysis step . . . 45
4.2 Relative total and prolog overheads of the code regions in % of the pure time. . . . 46
4.3 Properties found by the SCA analysis strategy supported by the TOR and POR instrumentation adaptation strategies. . . . 46
4.4 Phase time and the number of high-overhead regions removed for the analysis runs with no instrumentation strategy, TOR strategy, POR strategy and a reference run without any instrumentation. . . . 47
4.5 Properties and their severities found by the SCABF analysis strategy in PEPC with and without instrumentation strategies. . . . 48

7.1 Properties detected in iterations interval [1-64] of the CX3D progress loop. 98
7.2 Properties detected in iterations interval [65-128] of the CX3D progress loop. 99
7.3 Final set of properties reported for the CX3D application. . . . 99
7.4 Properties reported for the 129.tera_tf application . . . 104
7.5 Properties reported for the PEPC application . . . 108
7.6 Convergence phases detected for adaptation strategy 1. . . . 112
7.7 Convergence phases from adaptation strategy 1 converted to the time steps of the adaptation strategy 2. . . . 113

Chapter 1

Introduction

A hundred billion neurons interconnected by a hundred thousand billion synapses, the extreme complexity of the human brain makes it one of the most challenging subjects to study. At the time this thesis is written the neuroscience is still not able to deliver a precise understanding of how the brain works. Experiments are needed to unveil the hidden mechanisms, the experiments that are difficult or even impossible to perform using tissue samples or living animals. A computer simulation of the brain currently developed in the Human Brain Project [5] will make such experiments possible. It is estimated that such a simulation has to run at the speed of a quintillion, 10^{18}, operations per second. Development of both software and hardware capable of achieving such computational performance is the core activity of *High Performance Computing*, for short, HPC.

The current 10 fastest HPC systems, typically referred as supercomputers, [10] are still, however, in the peta-scale range, i.e. 10^{15} operations per second, which is a factor of 1000 below the target above. Nevertheless, even at this scale the performance comes at the costs of massive parallelism, customized and sophisticated system design and a power budget measured in megawatts. Application development complexity for these architectures scales proportionally making programming anything but easy.

A particular aspect of application development is traditionally and naturally an important concern for the HPC field, namely, *performance optimization*. In its simplest interpretation performance optimization typically strives for more simulation work computed in less time. For the scientist interested in the results of the simulation it means less time waiting for the experiment data or more detailed simulation results within the reasonable time. For the system provider a more efficient usage of the resources results in a higher utilization of the system and cost savings.

As the performance of any system has a theoretical maximum, the optimization process requires input knowledge about existing inefficiencies in order to transform optimization potential encapsulated in them into performance gains. Providing this knowledge is the task of *performance analysis*.

In many cases performance analysis relies on an empirical evaluation of execution performance when run on an HPC system. Taking into account the parallelism and the complexity of the supercomputers, the amount of generated empirical data is overwhelming. This makes manual investigation prohibitively tedious and arguments for the *tools-aided performance analysis* process.

Performance analysis with tools typically involves instrumentation, measurement collection and analysis steps. During the instrumentation the application being analyzed is augmented by inserting probe-functions capable of registering relevant execution events. During the runtime the events generated by the probe-functions are processed by the measurement tool which results in raw performance measurements. The knowledge about optimization potential is then obtained during the analysis phase either by interpretation of the results by the user based on visualization or through an automatic evaluation.

The picture becomes, however, very complex, when realized that the performance characteristics might change over the course of the execution. This concern rises its importance with the increase of the application runtime which can go up to month of non-stop simulation. In such cases, *performance dynamics* is an essential phenomena allowing to identify performance optimization potentials that are missed should the information be ignored. An interesting case study [57] describing an MPI communication performance bottleneck migrating from one process to another and degrading with runtime supports the statement above. With applications and hardware becoming more complex and, in particular, more dynamic, such effects will become even more widespread and paramount.

Temporal dynamics adds additional complexity level to performance analysis. Tools are highly valuable in this case, however, have to face challenges themselves. Adding the temporal dimension leads to linear increase in overheads, size of measurements, analysis complexity and visualization volumes. In this thesis we present concepts and their implementation in the context of the Periscope performance analysis tool allowing valuable insight in performance dynamics and at the same time mitigating the above challenges.

The rest of the thesis is composed as follows: the reminder of this chapter states the contributions of this work and gives an overviews to the Periscope performance analysis tool which was a context for the study. In Chapter 2 we give an introduction to the tools-aided performance engineering. Here we introduce main concepts, techniques and tools for performance analysis. Chapter 3 overviews the state-of-the-art techniques addressing the issue of temporal performance measurements and analysis. In the next chapter we propose a concept for automatic adaptation of instrumentation as well as its implementation and detailed evaluation. Chapter 5 presents a novel technique for an online collection and analysis of dynamic profiles allowing time-dimension-scalable processing of temporal performance data. In the next chapter we propose a number of algorithms for evaluation of performance dynamics as well as an automatic analysis strategy. The evaluation of the two techniques with four real-world applications is given in Chapter 7. Finally, chapter 8 provides a summary and a future work outlook.

1.1 Contribution of This Work

This work presents new techniques for low-overhead, time-dimension-scalable and automatic performance analysis of dynamic performance properties of HPC applications.

For the first time in the field advanced signal processing algorithms are applied in order to automatically evaluate dynamic properties of performance bottlenecks. These are used to quantify the amount of variability in time-series of performance measurements; to analyze dynamic properties at different time intervals and scales; to detect and characterize degradation trends both quantitatively and qualitatively. The results reduce the analysis efforts by the user by detailing the temporal location, severity and high-level qualitative description of the relevant performance degradation patterns. Alternatively, the results can be used as an input to a visualization tool to shrink the visualization volume by limiting it to the relevant time intervals.

Furthermore, the thesis presents a novel scheme for an efficient collection and analysis of series of dynamic profiles. It allows to mitigate the issue of linearly growing data sizes with analysis time. We achieve this in two steps. First, we extend the dynamic profiling technique with an online retrieval of samples for the remote analysis. This guarantees that the amount of measurements buffered on the monitoring side, i.e. the processes where the application is actually running, is invariant to the time dimension. Second, the time-series of measurements accumulated on the remote analysis side are automatically evaluated using the techniques described above in bursts of predefined length. It allows the analysis procedure to be scalable along the time dimension as well. Here the size of measurements stored is a function of the burst length and not the time. Thanks to the choice of multi-scale algorithms, the resulting dynamic properties can be merged together over multiple bursts.

We solve the problem of overheads mentioned above by developing automatic instrumentation adaptation strategies. The uniqueness of the presented approach is in enhancing source-level instrumentation with an ability to adapt to both generated overheads and online measurement requirements. This allows to combine efficient overhead reduction with a better granularity of instrumentation and a wider portability when compared to other approaches based on dynamic binary re-writing. The concept offers two strategies for trading-off measurement granularity versus introduced overheads. In the first strategy the introduced overheads are automatically evaluated and limited according to a given threshold. In the second case the overheads are reduced by minimizing the instrumentation to the absolute minimum needed to fulfill the requirements of the current state of the online measurement process.

The aforementioned concepts were implemented in the context of the Periscope performance analysis tool. We holistically improve the tool's capabilities in respect to time dimension in all the three major aspects of performance analysis: instrumentation, measurements, analysis. Moreover, the implementation allows to resolve critical limitations

and flaws in the analysis model of the tool vulnerable to performance dynamics.

1.2 Study context: Periscope

The context for this work is the Periscope [33], [34] performance analysis tool developed at the Technische Universität München. The main purpose of the tool is to automatically detect and quantify performance inefficiencies suffered by applications when running in large scale HPC systems. A short introduction to the main concepts of Periscope is given below.

1.2.1 Analysis Automation

A distinctive feature of Periscope is the automatic search for performance inefficiencies. The automation is achieved through formalization of typical performance inefficiencies in terms of performance properties. An example property could be "Excessive MPI wait time in receive due to late sender".

In order to test for the presence of an inefficiency, each property contains a condition expression. The degree of certainty in a found property is estimated using the confidence expression. Finally, each property quantifies the negative impact on the overall performance by means of the severity expression. This is an important value allowing to prioritize multiple performance properties and, therefore, guide the tuning process.

In addition to a high-level qualitative description of the problem, the property specification also carries the information about the property context. It is composed of a source code location specified by a file name and a line number, as well as, an execution location, which is an MPI process rank and an OpenMP thread number.

Searching for a property is equivalent to testing a hypothesis about the property holding in a specific context. Since the number of hypotheses to test is the cross product of the set of available properties and the set of contexts, sophisticated analysis strategies are used to perform the search efficiently.

1.2.2 Online Analysis

The analysis model of Periscope is an iterative online process when the measurements are configured, obtained, and analyzed on the fly, while the application is running.

The model is presented in Figure 1.1. It resembles the typical process of the closed-loop iterative hypothesis driven analysis. It starts with a root hypothesis stating the presence of the root performance issue within a set of contexts. Experimental data is needed in order to accept or decline the hypothesis. In order to obtain it, a performance experiment

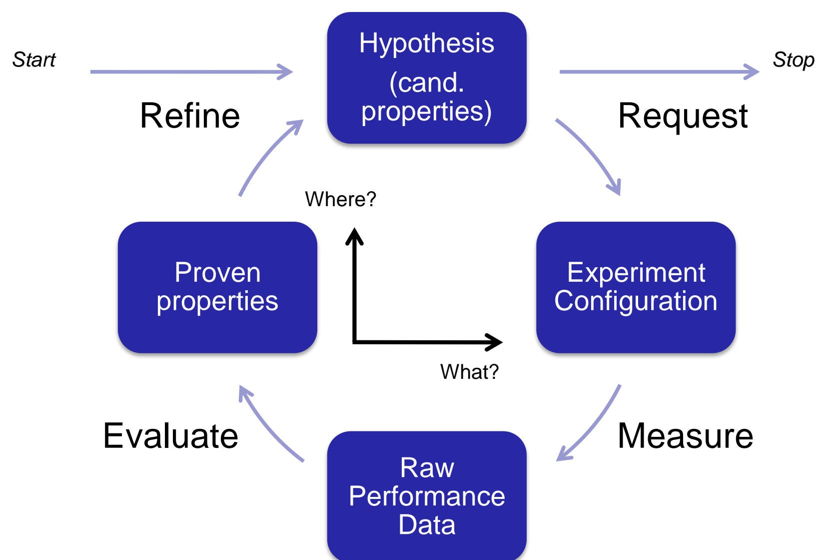


Figure 1.1: A schematic illustration of the Periscope analysis model.

is configured by sending measurement requests to the remote monitoring library linked to the application. The experiment is then carried out by measuring one *phase* of the application. The phase typically corresponds to one iteration of the main loop. The resulting raw performance data is then returned to the analysis agent for evaluation of the hypothesis. Based on the outcome, the accepted hypothesis (i.e. proven properties) is refined into a new set of hypotheses. During the refinement process Periscope explores the space of the potential performance issues by trying to narrow down the location of the problem, i.e. answering the “where?” question, as well as to precisely identify the type of the issue, therefore, answering the “what?” question. The cycle is then repeated, unless all the hypotheses are evaluated.

1.2.3 Distributed Analysis

A simplified diagram of the Periscope architecture is shown in the Figure 1.2. The tool is composed of multiple hierarchically distributed agents which are functionally categorized into the layers shown in the figure.

The top functional layer is composed of the GUI, which is a part of the Eclipse integrated development environment. This allows for a closer integration between the application development and the performance analysis processes.

The next layer is represented by the Periscope frontend agent. This is the agent that is responsible for the start-up and global control of the online analysis process.

The reduction layer is represented by a n-ary tree of command propagation and results

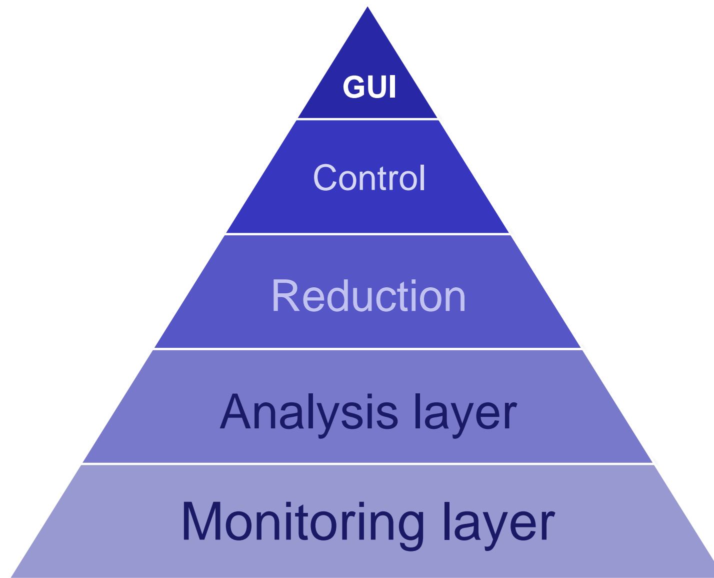


Figure 1.2: A simplified schematic illustration of the Periscope architecture.

aggregation agents. The leaves of this tree form the analysis layer which is composed of the agents responsible for carrying out an automatic iterative analysis in a subset of application processes.

At the bottom of the hierarchy are the application processes linked with the monitoring library. Currently Periscope supports two libraries: an internally developed MRIMonitor as well as Score-P - the joint measurement infrastructure used by a number of other tools.

The number of agents at each level is not constant and scales together with the number of application processes. This flexibility in size and configuration of the distributed architecture guarantees scalability of the analysis process.

1.2.4 Current Limitations

The design of Periscope allows to efficiently address most of the challenges in the performance analysis of HPC applications. Nevertheless, Periscope completely disregards the temporal dimension of performance. Moreover, temporal performance variability leads to incorrect analysis results. Additionally, the valuable insights about performance dynamics cannot be produced within the original design. This makes Periscope an interesting context for developing and testing new technologies making performance dynamics analysis possible.

Chapter 2

Tools-Aided Performance Engineering

The complexity of HPC applications, on one side, together with the complexity and massive parallelism of HPC systems, on the other side, argue for a structured and methodical approach to the performance optimization process. Here various tools play an important role in making it less tedious, allowing otherwise missing insights or even automating parts of it.

In this chapter we give an overview of the tools-aided performance engineering process with a focus on performance analysis tools. We discuss the major tool technologies and briefly introduce several prominent examples.

2.1 Software Development Life-Cycle

As a part of the application development process, performance optimization activity has to be put in the context of a software development life-cycle. According to the IEEE Standard for Software Maintenance [12] and its revision in 2006 [13] performance optimization is one of the activities of the *maintenance phase*. The maintenance phase is entered after the software is released and is typically the longest phase of the development. In addition to performance optimization activities, the phase also covers repairing defects, i.e. fixing bugs, and porting the software to new environments.

2.2 Performance Tuning Cycle

Although the tuning can be performed without a well-defined process, which is, unfortunately, quite often the case, it is certainly not the most productive approach. Figure

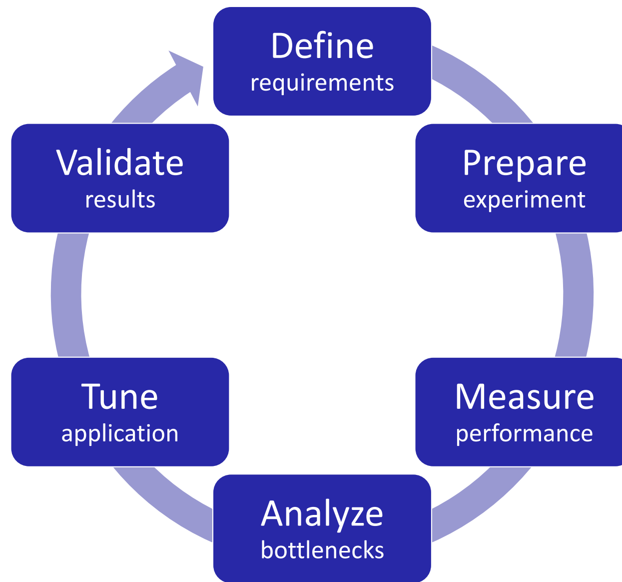


Figure 2.1: Performance Tuning Cycle Methodology.

2.1 presents a high-level diagram of a cyclic tuning process offering a more structured workflow. It consists of the following steps:

Define requirements. Before jumping directly to modifying source code, it is highly recommended to specify optimization objectives. An example could be a reduction of the application execution time by a certain factor. Since the execution time of the same application can vary depending on the machine, data set, application or environment configuration used, it is also important to fix this parameters beforehand. When this step is entered after one cycle again, it is additionally a pivotal point when the decision whether to abort or continue tuning has to be taken. If the decision is to continue the requirements might have to be refined.

Prepare experiment. During this step a performance analysis experiment is prepared. It includes selecting and configuring a tool to be used for instrumentation and measurement of the application, instrumenting the application, configuring the system and preparing a reference for a later validation step.

Measure performance. During this steps the instrumented application is executed on the target HPC system. While running, performance measurements are collected by the tool. As the result of this step raw performance measurements are produced. The size, high-dimensionality and complexity of this data make the knowledge about performance inefficiencies obscure.

Analyze bottlenecks. The goal of this step is to discover the potential for performance improvement (in terms of the optimization objective) contained in present bottlenecks. This is achieved by interpretation of the raw performance data collected at the previous step by computing derived metrics, visualization techniques or even automatic analysis. By searching for inefficiencies one has to answer the following questions: "Where?" - location in source code and in system topology; "What?" - detailed understanding of the problem type; and "When?" - location in the time domain. Importantly, one has to quantify the severity of the problem which gives an estimation for potential optimization gains.

Tune application. Based on the detailed information about performance inefficiencies, one can take an informed tuning decision on choosing the targets for optimization. However, a specific tuning action to be performed is often not trivial and is typically produced by the application developer. An example could be selecting proper compiler flags or even completely replacing some poor-performing algorithms. Nevertheless, the tools start to arise that allow for automatic tuning of certain aspects of performance.

Validate results. After the tuning action was implemented, one has to make sure that the tuning has not corrupted the application logic and that the results are still correct. It is also at this point when the achieved performance improvement is compared against the acceptance criteria set in the beginning. In order to keep track of the applied analysis and tuning it is highly recommended to protocol completed steps and achieved results.

Motivated by ever growing complexity of the tuning process, various tools have emerged. Some support users in one of the aforementioned steps, some target to automate parts or even the complete workflow. There is also a broad spectrum of technologies differentiating the way tools perform similar tasks. In the following writing we discuss the major technologies and provide some examples of tools. We give the overview under the angle of performance dynamics.

2.3 Monitoring Performance

As any other observation, the process of performance monitoring requires active acquisition of information. This means that the execution of the application has to be interrupted and the state of the execution together with other relevant information recorded. Based on the way the interruptions are performed and the way obtained information is processed and stored one can differentiate the main techniques of performance monitoring.

2.3.1 Sampling

Sampling is a technique using asynchronous (in respect to the program execution) interruptions of an application in order to obtain statistical information about performance. Interruptions can be done regularly with a certain frequency, or irregularly, based on external events. In the second case the interruption event is typically an overflow of a hardware counter.

After the interrupt was triggered the monitoring library function is called. It accesses the currently executed instruction by looking at the program counter. The monitoring library then increments the sample counter associated with the interrupted instruction. Additionally, other metrics such as encountered cache misses or completed floating point operations can be recorded.

The advantage of the technique is that the overhead introduced by sampling is typically low and proportional to the frequency of interrupts which can be easily controlled. On the other hand, the measurements collected this way are distributions of samples over instructions, which might miss a rare but an important event. It also fails to record the information about dependencies between the events such as parent-child.

2.3.2 Direct Instrumentation

The limitations of the sampling approach can be overcome with the direct instrumentation, however, at the cost of higher and less manageable overheads.

In this case the interruptions are synchronous and are triggered internally by probe functions inserted in the application code. This approach allows a very precise measurements since *every* instance of the event is intercepted and processed. Additionally, a valuable information about the order of events is captured.

Based on the way the probe functions are inserted one can distinguish the following direct instrumentation techniques:

Source Code Instrumentation

As it follows from the name, in this case instrumentation is inserted into the source codes before the compilation happens. This can be done manually by the user or automatically by a source modification tool or a compiler (often referred as compiler instrumentation).

This approach features several important advantages. First, it allows a granularity down to a line number. Second, the approach offers maximum flexibility in selecting code regions to be instrumented. And finally, the mapping between the source code entities and the generated measurements is straightforward.

The disadvantages is the dependency on the availability and the programming language of

the source code. Additionally, re-compilation is unavoidable should the instrumentation be changed.

This approach is taken in Periscope [18] but also by Tau [56] and the OPARI [46] instrumenter. Within the Tau project a generic source level instrumenter was developed that can be used by other tools to insert their own instrumentation code in FORTRAN and C applications. OPARI is a source level instrumenter for OpenMP. It allows to instrument all OpenMP regions in FORTRAN and C codes.

Object Re-writing

Another approach would be to insert probe functions directly into the machine code of a compiled application. This way the application can be instrumented even when the source code is not available. Moreover, the machine code can be modified in on-line mode. This adds a capability to modify instrumentation without recompiling and restarting the application.

The disadvantages are limited portability, coarse function-level granularity and sophisticated mapping of measurements back to source code.

This approach is followed by two widely known tools: DynInst [3] and DPCL [24].

Library Interposition

This approach is based on creating a wrapper function around a library function. Monitoring of commonly used library functions such as MPI functions or memory allocation functions is performed this way. In case of MPI functions weak symbol mechanism is used to transparently substitute the library symbol with a wrapper symbol at link time. The actual library function is then called from within the wrapper using the `P` prefix. Instrumentation is placed in the wrapper before and after calling the original function.

2.3.3 Measurement Overhead

The instrumentation techniques discussed above require interruption of application, thus the consequence are overheads. One can break down negative effects of overheads into the following categories:

- Prolongation of the execution time
- Artificial load-imbalance
- Alternated cache behavior

From this list, the first two are the most common and severe problems. The typical reason for the prolonged execution time is the instrumentation of a relatively "tiny" but frequently executed function, e.g. C++ getter function. In this case the overheads of calling the probe function are comparable to the amount of the time spent in the function itself. Since the function is frequently executed the overhead times sum up into a significant slow-down, which can be in orders of magnitude.

The artificial load-imbalance problem arises when the amount of overheads varies between the tightly communicating processes. Additional overhead in one process will lead to waiting time in the other one. An example here could be asynchronous flashing of measurement buffers to the discs when the allocated memory is exhausted.

In the context of analysis of performance dynamics in long running applications overheads are an important concern. First of all the amount of overheads (though in absolute numbers) linearly grows with the analysis time. The consequences are twofold: first, the analysis time is extended proportionally; second, the risk of measurements corruption increases due to non-linear effects, such as a trace flush. The last factor is particularly important, taking into account potential loss of significant computational resources invested in a corrupted and therefore useless measurements.

2.3.4 Mitigation of Overheads

In order to reduce the risks of corrupted measurements data, overheads have to be mitigated. Mitigating overheads in general means choosing the acceptable balance between instrumentation granularity, dictated by measurement requirements, and overheads. This, however, requires the knowledge about the both parts of the equation. Additionally, measurement requirements are not static during the analysis and might change during the on-line analysis process (for more details see Section 2.4.2).

So what are the alternatives for mitigating overheads? The alternatives are either to compensate the already introduced overheads or to decrease it. The first case requires a model of the overheads and its influences on the application. Taking into account that overheads impact application performance non-linearly, like in case of artificial load-imbalance discussed in the previous session, developing such a model is very complicated.

Therefore, most of the techniques follow the second approach of reducing the overheads. Here we again have two options: reduce the amount of overhead caused by one interrupt, or to reduce the number of interrupts.

In the first case the goal is to make the probe-functions as light-weight as possible but still capable of capturing the necessary information. In this trade-off the two main approaches to processing the measurement data have emerged: profiling and tracing. We discuss the two techniques in more details in Section 2.4.2. Nevertheless, the overhead caused by one interrupt cannot be completely removed, therefore the total impact can be only decreased by a certain constant factor.

The second option of reducing the number of interrupts but still being able to capture the relevant information is more promising in this respect.

In case of sampling this is achieved easily by reducing the frequency of interrupts. The relation between the frequency and the resolution of measurements is also straightforward.

However, in case of direct instrumentation this is not that trivial. Here the number of interrupts is defined by the number of source code locations instrumented and the number of times each location is executed. Since the number of executions is dictated by the application logic, this part of the equation cannot be affected. So, we are left with the first option. The main overhead mitigation techniques utilize this approach.

The main question here to answer is which source code locations have to be instrumented and which not? The answer to this question is the intersection of two sets: the locations relevant in terms of the analysis and the locations which generate less overhead when instrumented.

There are different strategies to answer this question. Tools like Score-P answer this question by estimating the overheads first by performing a light-weight performance measurement, and then filtering out (though, not completely eliminating) the instrumentation of locations that are responsible for most of the overhead and leaving everything else. The estimation here is based on the number of times the location is executed. Therefore relatively short but frequently executed functions are eliminated from the more detailed follow up analysis.

It is important to mention that the knowledge about this tiny functions is already a valuable insight. Even a slight improvement to the performance of this functions, or ideally complete elimination, might result in significant speed-ups.

The ParaDyn tools takes a different approach of dynamically instrumenting only those locations which are of interest at the current state of the on-line analysis. After the analysis of the location is completed the tool removes instrumentation. This approach allows to reduce the overhead to the absolute necessary minimum.

2.4 Performance Measurement

After the necessary events are captured by the instrumentation relevant information has to be processed by the measurement tool and stored away for a consequent analysis. Based on **what** is stored and **when** it is analyzed there are two dimensions to categorize the tools.

2.4.1 Profiling vs Tracing

There are two main approaches to answering the “what?” question.

In the first case, called **profiling**, the structural entities of application are mapped onto aggregated values of measurements. The aggregation can be done over multiple dimensions. In the most typical case the aggregation is done over the time dimension. Also other types of aggregation are possible, e.g. processes, threads. The structural entities of application are determined by the instrumentation applied and are called *regions*. Based on the type of interrupt processing applied one can distinguish *flat* and *call-path* profiles. In the last case the parent-child relation between the regions is preserved and the entities are the unique sequences of regions where a parent region calls a child. The advantage of this approach is that the amount of measurements generated is relatively low. This comes at the price of losing the information due to the aggregation procedure.

An extension of the profiling allowing to capture the temporal evolution is called **dynamic profiling**. It exploits the iterative nature of many scientific applications to create a time series of profiles, where each sample is a profile of one iteration of the progress loop. Though, the size of the dynamic profile per iteration is low, it grows linearly with time and therefore not scalable in case of long running applications.

Alternatively, **tracing** allows to process every event captured by the instrumentation. Every time an interrupt occurs, a time-stamp is read from a timer. The event is then stored in to the trace buffer together with the time-stamp and the relevant context information. This can be, for example, the message size or the destination rank of a `MPI_send` function. Traces allow full sequence of events at the cost of linearly growing with time trace size.

2.4.2 Post-mortem vs Online Analysis

There are two alternatives to answering the question of when to analyze the collected measurements.

Post-mortem analysis, as it follows from the name, it is delayed after the end of the application run. In this case the performance measurements are collected and stored into internal buffers (sometimes flushes to the disk have to be done when the buffers are exhausted) while the application is running and then dumped to the disk. The data is then read and either visualized for a manual investigation or automatically analyzed by the analysis tools. The advantage of this approach is that at the analysis time all the data is already available so a complete analysis can be guaranteed.

Online analysis, alternatively, is happening while the application is running. Typically it is performed in batches, where the application is monitored for a period of time and then the collected measurements are immediately evaluated by the tool. The results, which are typically less dimensional, are kept and the raw performance data is discarded. After the analysis of the batch is completed, it can be either aborted or refined. Here lies one of the main advantages of the online analysis. It does not require a complete run in order to produce valuable results, or can collect more information within one run, when some parts of this information cannot be obtained within a single run. It also allows to tremendously

reduce the amount of output information. The disadvantages of this approach are that the online analysis will cause additional overheads. This problem, though, can be solved by moving analysis process to a remote location. Also, the online analysis is performed only on a subset of measurements, which limits the scope.

2.5 Performance Analysis Automation

The ultimate goal of performance analysis is the knowledge about the performance improvement potential. This is, however, not immediately available from the raw performance data collected during the measurements. One needs to perform analysis step in order to extract it.

Due to the variety and complexity of the data and the underlying performance phenomena there is no standardized analysis procedure. There are, however, some steps which can be automated. The degree of analysis automation is, therefore, another important characteristic of performance analysis tools. The two extremes here are a solely manual analysis, when the raw data is presented to the user, and a completely automated approach, when the precise information about the detected improvement potential is presented. As usual the extremes are either impractical or impossible. Therefore a combination of a manual analysis by the user supported through an efficient automation is the most promising approach.

Many tools support automatic computation of relevant metrics derived from raw data. In addition to that, Scalasca [42] automatically searches traces for predefined inefficient communication patters. The data is then presented for the interactive exploration using the CUBE [30] browser. Tools like ParaDyn [45] and Periscope go one step further and automatically search and quantify predefined and formalized inefficiencies.

In essence, automation allows achieving the following benefits:

- Improved time to solution
- Less error-prone
- Compensates missing in-depth knowledge
- Enables analysis of large-scale real-world applications

2.6 Tools for Performance Engineering

In this section we give an overview of a number of tools, where each represents one of the typical tool's classes. These are differentiated by their purpose as well as the used

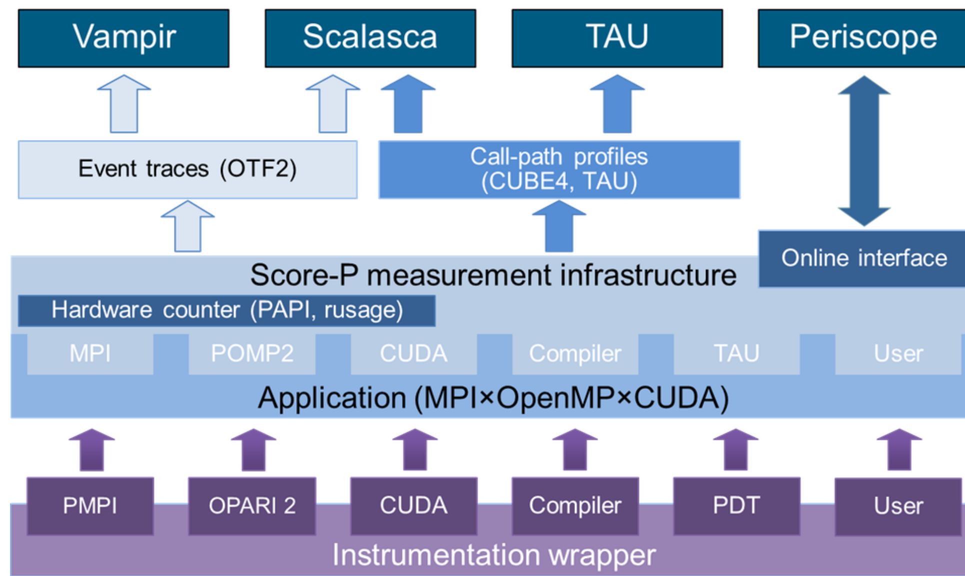


Figure 2.2: Score-P architecture[8]

combination of technologies described above. In the following writing we will discuss in more details the following tools:

- Score-P - Multi-mode instrumentation and measurement infrastructure
- Scalasca - Post-mortem trace-based automatic analysis
- Vampir - Post-mortem visualization trace-based manual analysis
- Paradyn - Online automatic profile-based analysis
- Pathway - Performance optimization workflow automation and execution

2.6.1 Score-P

Score-P [38],[14] is a joint instrumentation and measurement infrastructure supporting most of the technologies described above. The motivation behind developing Score-P was to create a common measurement platform for a number of analysis tools, namely Periscope, Scalasca, Vampir and TAU. Supporting all the instrumentation and measurement techniques described in the previous section, the tool additionally offers higher quality, scalability and portability.

High-level architecture of the tool is shown on Figure 2.2. Score-P consists of an instrumenter, showed in the lower part of the picture, a measurement library linked to the application, shown in the middle, and a number of output interfaces and data formats

enabling an efficient consumption of measurements by the analysis tools, shown in the upper part.

The tool supports both direct and indirect, i.e. sampling, instrumentation modes. The direct instrumentation can be done at the source, library and binary levels. The performance aspects that can be instrumented are MPI calls, OpenMP constructs, CUDA kernels, function calls as well as user defined source code regions. The instrumentation functions inserted in the application resolve to the corresponding Score-P measurement adapter functions. These translate the instrumentation calls to a set of events which are processed by the measurement core. Here additional information such as hardware counters or timings are read and associated to the event. This information is then consumed, depending on the given Score-P measurement configuration, by the call-path profile module or the tracing module. After the application terminates the collected profile data is stored in the CUBE4[31] profile format and the trace is stored in the OTF2[25] trace format. Alternatively, on-line consumption of the measurements is supported by means of the Online Access Interface. This allows remote configuration, extraction and execution control over sockets.

2.6.2 Scalasca

Scalasca [42] is an open source scalable trace-based automatic performance analysis tool developed at Forschungszentrum Jülich (FZJ). The main purpose of the tool is to support the user in identifying communication and synchronization bottlenecks in MPI, OpenMP and hybrid applications.

Scalasca performs post-mortem automatic analysis of traces. It relies on Score-P measurement infrastructure for instrumenting, monitoring and producing traces of an application. The results of the analysis are visualized and interactively explored using a browser called CUBE.

The analysis algorithm of Scalasca automatically searches and localizes wait states. These situations are very typical inefficiencies encountered in parallel applications. A wait state is suffered by an application process that needs some data from another application process in order to continue with the computations. When the data is not yet available because the other process is being late with delivering it, the first process has to pause the execution and enters the wait state.

Since the size of the trace and thus the analysis effort grows depending on the number of application processes, Scalasca employs parallel trace replay mechanism for the identification of wait states. After the monitoring of the application is finished and the trace is written to the disk, Scalasca trace analyzer is started with exactly the same amount of application processes. In the beginning, each process of the analyzer loads the trace of the respective application process. After this is done, the forward replay of the recorded communication is initiated. During the replay each analyzer process repeats the com-

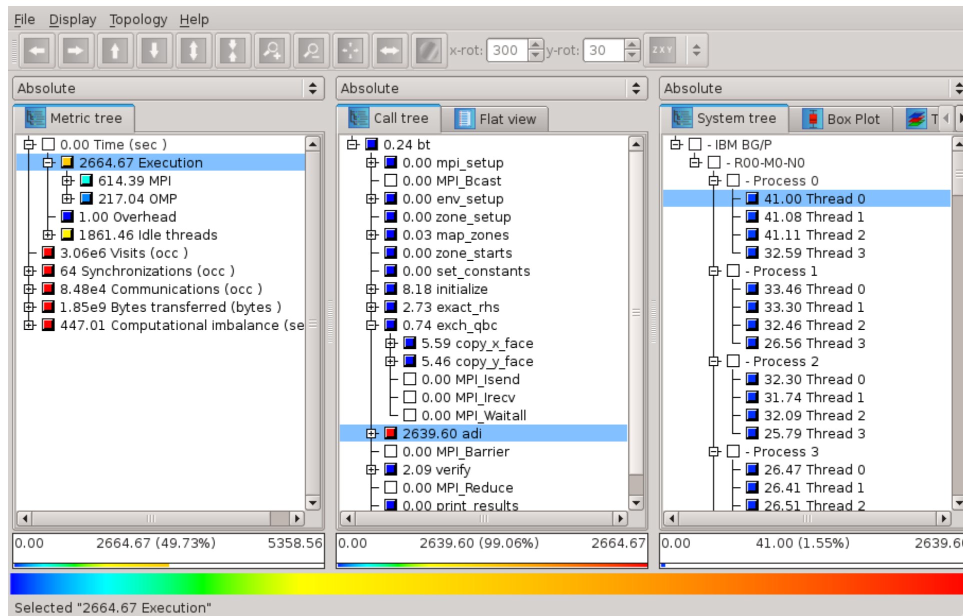


Figure 2.3: CUBE GUI[7]

munication operations performed by the application, but instead of the original payload the time-stamps are exchanged. Having this information the wait states are computed by taking the difference of the time-stamps. Since the trace contains also the information about call-paths and processes, the wait states can be properly attributed to the locations where they were encountered. The scalability of the approach was proven with experiments with up to 294,912 processes.

Figure 2.3 shows the CUBE [30] interactive performance browser which serves as a graphical user interface for Scalasca. The data model of CUBE consists of three dimensions: metrics, program (i.e. call-paths) and system (i.e. processes, threads). Each dimension is represented as a hierarchy allowing different levels of granularity. An example of a hierarchy in the metric dimension could be a parent child relation between a metric MPI communication time which is included in the metric Total Execution Time. A severity function defines a mapping of the entities (metric m, call-path c, process p) of the three dimensions onto the accumulated value of metric m. The dimensions are represented by three tree-browsers, representing from left to right the metric, the program and the system dimensions. Two actions are available to the user: selecting a node and collapsing/expanding a node. At each point in time, there are two nodes selected: one node in the metric tree and a node in the program tree. Selection of a node in the system tree is not supported. Each node is labeled with a severity value which is color-coded. A typical use case is to read the tree-views from left to right. A value shown in the metric tree is a sum of a particular metric over all call-paths and system entities. A value shown next to the node (i.e. a particular call-path) in a program tree is a sum of the values of the selected metric over all processes. And finally, the value shown next to a node in the system tree

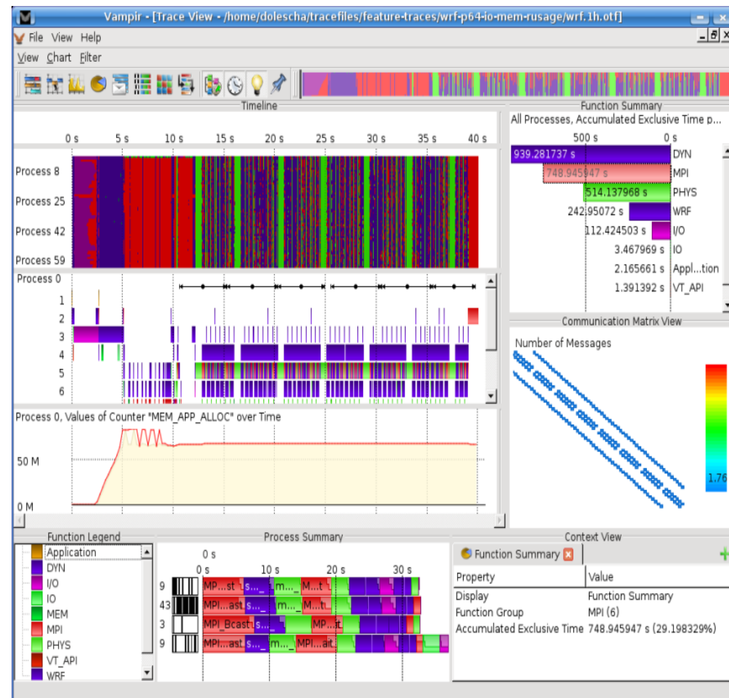


Figure 2.4: Vampir GUI[11]

is a value of the selected metric measured in the selected call-path on a particular process. All values could be either absolute or percentages of a maximum value.

2.6.3 Vampir

Vampir [47] is a post-mortem trace visualization tool for a manual interactive investigation of the application performance. Vampir uses Score-P in order to instrument, monitor and produce traces of an application execution.

The main purpose of the tool is to translate events and corresponding measurements contained in the trace into various graphical representations. These allow detailed insights in various aspects of the application performance by means of powerful user-interaction functionality. Additionally, the tool offers cross-experiment comparative analysis. Providing maximum possible level of details Vampir enables the user to identify performance optimization potentials which are often missed by other tools.

Vampir GUI shown on Figure 2.4 offers two types of displays: time-line and chart views. The first one presents events as a chain of color- coded blocks and supporting symbols over an arbitrary interval of the time dimension. Charts, alternatively, show summarized values of various performance metrics computed from the events contained in the trace. Vampir GUI can be configured to display multiple views, both time-line and charts, within one window. What is important is that all the currently visible views are coupled. This

means that when the user changes one view configuration, e.g. zooming or scrolling displayed time interval, Vampir automatically adjusts other views.

Vampir is a powerful exploration tool offering full information about the recorded performance behavior. When the right "angle of view" is selected performance inefficiencies clearly stand out, often with the eureka effect. However choosing the "right angle" is not straightforward, especially in case of long running applications with dynamic performance.

2.6.4 ParaDyn

ParaDyn [45] is an online automatic performance analysis tool. It employs automatic dynamic binary instrumentation for inserting and removing probe functions only in those locations and time intervals that are relevant for the online analysis process. The instrumentation infrastructure of ParaDyn, called DynInst, is available as a standalone tool and is also used in a number of other tools, such as VampirTrace [47] Open SpeedShop[54], TAU [56] to name few.

The tool also employs a distributed architecture consisting of the main Paradyn process and one or more Paradyn daemons. The main process is responsible for performing data management, automatic analysis process, visualization and user interface. The daemons perform instrumentation, measurement collection, measurement delivery and process control of a subset of application processes. When running in large-scale HPC systems, Paradyn uses MRnet [53] for a scalable data reduction.

The Paradyn process responsible for the automatic analysis is called Performance Consultant (PC). PC utilizes the W^3 analysis model, where the search for performance bottlenecks is performed along the three "W" dimensions depicted by the pronouns: "Why?", "Where?" and "When?".

The first "Why?" dimension defines the space of potential performance problems which can be encountered in an application. Each point in this dimension is represented by a hypothesis stating a particular problem and a condition expression used to prove or disprove the hypothesis. The hypothesis encode general problems and thus are independent of the program and the algorithms being analyzed. The hypotheses are organized hierarchically, which allows an efficient way of evaluating the space of hypotheses by going down from the root to the leaf hypotheses.

The "Where?" dimension covers all the execution locations where a hypothesis is evaluated. These are composed of application regions, machine resources, parallel programming language constructs. The components are typically organized in hierarchies of entities and can be statically defined as well as added at runtime dynamically. For example code locations are defined by a module, which in its turn is a collection of procedures. Therefore a particular performance can be first checked for the whole module and then be refined to one of the procedures inside.

A distinctive feature of Paradyn is the "When?" dimension. This was one of the first

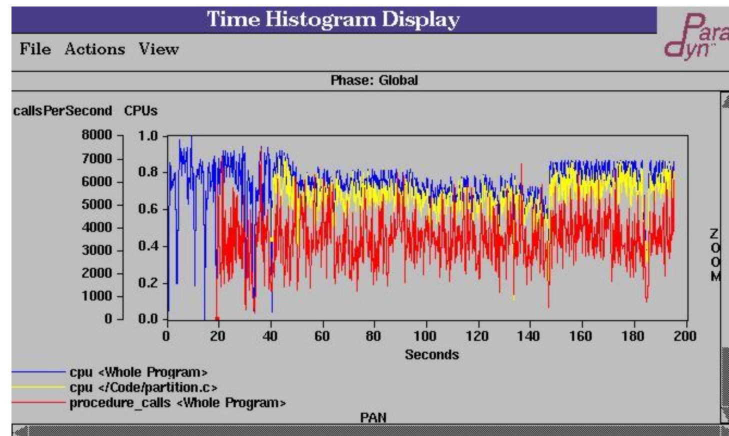


Figure 2.5: ParaDyn Time Histogram Display[6]

tools to tackle performance inefficiencies along the time dimension automatically. The goal of the analysis is to locate bottlenecks in multiple time intervals called execution phases. Search for bottlenecks is then performed independently and detected bottlenecks are attributed to the phase where they were found. Within each phase Paradyn records measurements into a time-series buffer. The size of the buffer is constant and is allocated for each phase regardless of the temporal duration of the phase. Naturally, this leads to the situations when the buffer space is sooner or later exhausted. Paradyn solves the problem by dynamically doubling sampling period and corresponding coarsening of the already recorded data. This leads to logarithmic decrease in sampling rate and loss of higher frequency details of the signal. The resulting signal still reflects important temporal behavior since the short term changes lose their significance for the analysis with the phase time growing. Recorded time-series can be then plotted for each phase independently and visually analyzed by the user. Figure 2.5 shows a time-series measurements recorded for some application.

The analysis is performed by the Performance Consultant (PC). It searches for performance bottlenecks by automatically evaluating and refining hypotheses following the W^3 model. During the refinement step a new set of possible refinements is generated following the hierarchy down from the hypotheses proved in the previous steps. The possible refinements are grouped into sets that can be evaluated at once, since all cannot be evaluated together due to monitoring limitations. Additionally, evaluation costs are estimated for each of the refinements which are ordered accordingly. This way a part of the refinements is selected for the evaluation and is proven or disproven based on the collected measurements. The whole process of the refinement can be done completely automatically or semi-automatically. The executed refinements and the results of the evaluation are then presented to the user inside the search history graph (SHG) shown in Figure 2.6.

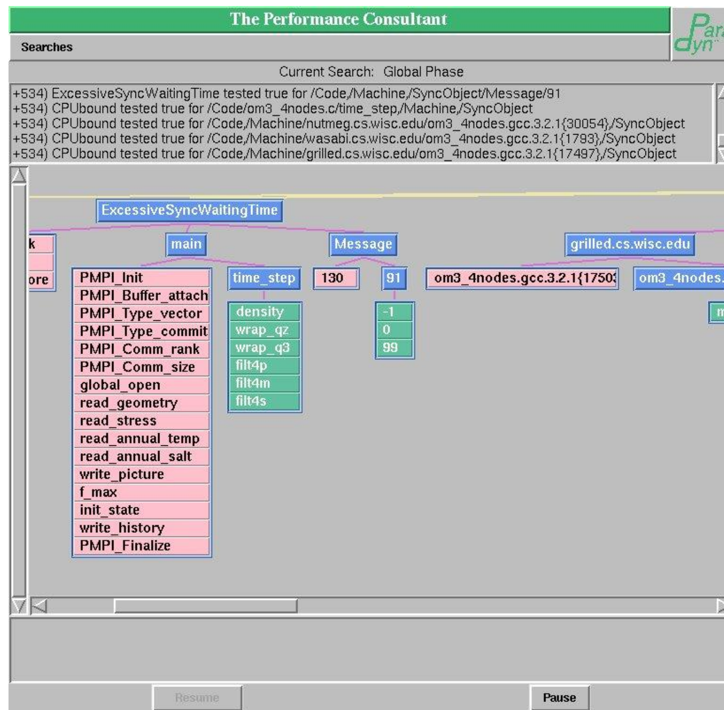


Figure 2.6: ParaDyn GUI[6]

2.6.5 Pathway

Pathway [52] developed at Technische Universität München is a tool that supports the user in structuring, executing and tracking the process of performance engineering. The tool is built around the notion of workflows defining typical steps and transitions between them which is borrowed from the field of business intelligence. The tools is implemented as an Eclipse plug-in and relies on a palette of technologies such as Business Process Model Notation (BPMN), Parallel Tools Platform, source code revision control as well as a number of performance analysis tools.

Workflow editor is a central part of Pathway and is used to construct, execute and monitor performance engineering processes such as the optimization cycle described above. Workflows are constructed using custom domain-specific nodes that formalize typical steps in performance analysis and optimization. The nodes are connected by arrows which indicate transitions between them.

Figure 2.7 shows the scalability analysis workflow implemented in Pathway. Scalability analysis is a common analysis procedure which requires multiple experiments with a varying number of processes. Often done manually, it is known to be a tedious process. Alternatively, Pathway completely automates this process by automatically instrumenting the application, configuring and executing a series of experiments on a remote HPC system, retrieving, storing and visualizing results using the tool of choice. The only input

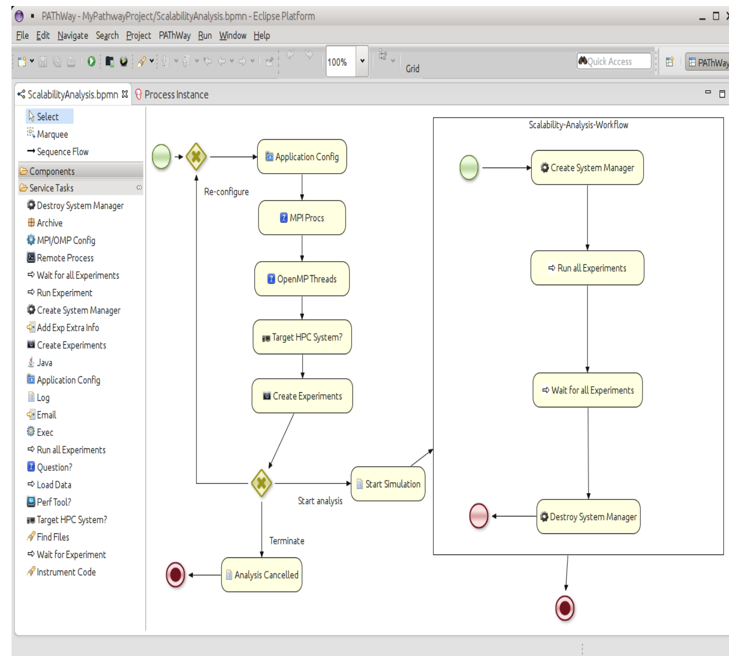


Figure 2.7: Pathway workflow editor

required from the user are the HPC platform, performance tool and the range of number of processes.

Another important feature of Pathway is a very detailed protocoling of the executed experiments. For each experiment executed Pathway automatically collects a large set of metadata. These include a snapshot of the application sources; HPC system configuration such as the values of environment variables and loaded modules; configuration of the applied performance analysis tool; as well as user comments provided at the beginning of the experiment. This data is a valuable information allowing to track performance engineering efforts over long running projects as well as ensuring reproducibility of results. The historical data of all executed experiments could be explored using the browser shown in Figure 2.8

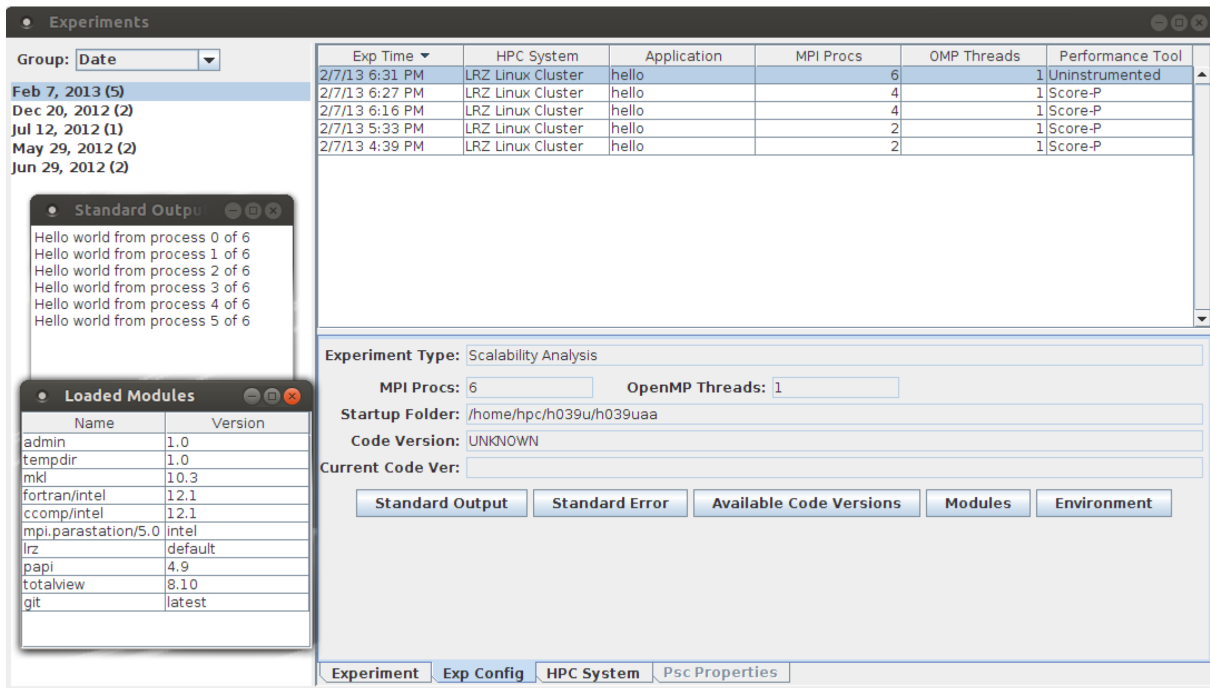


Figure 2.8: Pathway experiment browser

Chapter 3

State of the Art

In this chapter we present state of the art techniques addressing the temporal dimension in performance measurement and analysis.

3.1 Compression

The size of performance data linearly growing with time is an important concern to the most of performance analysis tools. Hence the number of techniques were developed to cope with the issue. Most of the techniques address the problem by compressing the data. In this section we present a number of promising compression techniques.

3.1.1 Clustering of Dynamic Profiles

A compression algorithm allowing to mitigate the problem of linearly growing size of the dynamic profiles is presented in [58]. The technique is based on an on-line incremental semantic clustering of dynamic call-path profiles. The lossy compression algorithm exploits the repetitive behavior of the application in order to group similar iteration profiles together.

The clustering is performed locally in each process. This allows to avoid overheads introduced by communication and synchronization required otherwise. With every new iteration of the progress loop, a new iteration call-path profile is created. This initially constitutes a new cluster. This process continues until the maximum number of clusters is achieved. After that, the two nearest clusters are merged. Each cluster stores mean metric values and the iterations of the individual elements aggregated in the cluster.

In order to calculate the similarity of two clusters, a distance function is used. Since calculating the distance function based on metric values of all call-paths in a profile iteration is inefficient, condensed metric values are used instead. These are obtained

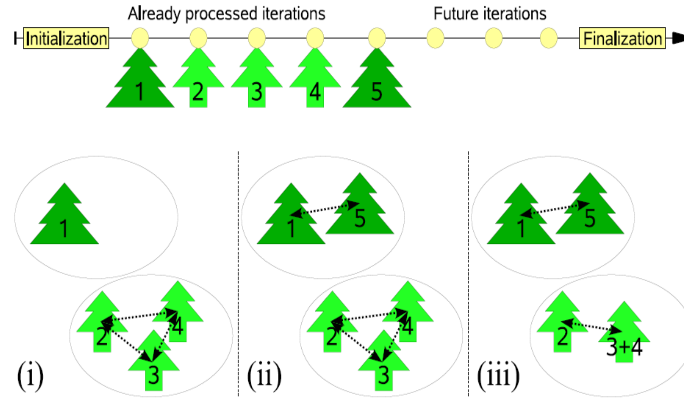


Figure 3.1: A schematic process of the incremental on-line clustering into a maximum of 4 cluster. (i) The first four call-path profiles are collected and categorized in two equivalence classes. (ii) The profile 5 is collected and categorized to the equivalence class of profile 1. (iii) Call-paths 3 and 4 are merged together as they are the closest ones according to the distance function. [59].

by aggregating them across the call-paths of the profile. Additionally, this allows to reduce the dimensionality, which leads to a better clustering quality. Manhattan distance operator is then used to capture the distance value.

However, the structure of the call-path profiles collected in two different iterations is not necessarily identical. In order to avoid introducing phantom branches or, oppositely, losing others, the grouping is only performed among the profiles with equivalent structure. Authors offer two ways do define equivalence:

- *Weak equivalence* - both profiles have identical call-paths
- *Strong equivalence* - both profiles have identical call-paths and the numbed of visits of those in both profiles is the same.

The schematic depiction of the algorithm is presented in Figure 3.1.

The presented algorithm provides an elegant solution to the problem of storing temporal performance profiles. It allows the user to control the size of the generated data by limiting it with the maximum number of clusters. Compression however comes at the price of lost information.

3.1.2 Wavelet Compression of Load Balance Measurements

A technique for scalable reduction of performance data volumes based on two-dimensional wavelet compression is presented in [28], [27]. The authors recognize that the size of col-

lected performance measurements is not scalable across processes and over time dimension and therefore has to be reduced.

The technique is based on a parallel, lossy, two dimensional wavelet compression [50] to gather performance data from all processes of a highly-parallel application. The dimensions are the processes and progress loop iterations, which can be also seen as execution time. The time required for the compression is low enough, which makes it possible to use the technique in real-time monitoring at scale.

At the end of the measurement collection, each process has a local vector of measurements. In order to perform the parallel transform the vectors are consolidated by a nearest-neighbor communication. During this step every S process receives S rows from the neighbor processes. The number of rows is selected in the way that it is large enough to half it L times, where L is the level of the Wavelet transform, and still be above the half width of the Wavelet filter. For more information on the Wavelet transform see Section 6.2. Afterwards, the wavelet transform is applied in parallel on each consolidated block of rows.

The resulting coefficients are then encoded using the Embedded Zerotree Wavelet (EZW) [55]. An important advantage of the algorithm is an efficient trade-off between the size and the accuracy. At the heart of the algorithm is an incremental thresholding using a successively smaller threshold. On each pass of the threshold one bit is stored in the output value indicating whether the coefficient is higher or lower than the threshold. Taking into account the compactness property of the wavelet transform the resulting encoding preserving the large coefficients is very space-efficient.

In the final stage, the locally EZW codded coefficients are merged with a parallel reduction. Huffman encoding [36] is then applied to the resulting buffer.

The authors demonstrate that the compression time is nearly invariant with system size and allows to preserve significant qualitative features of the data even for very large compression rates.

3.1.3 Compressed Complete Call Graphs

Another approach to compression based on the semantic information contained in the trace is called compressed Complete Call Graph (cCCG) [37] and is implemented in VampirNG [20].The approach is based on the graph data structure, called Complete Call Graph (CCG), which is used to hold the trace. In CCG every node is one instance of a function call storing the function metadata plus the time duration of the call. All function calls from within the given function will be added as children to the current node. Additional information such as MPI messages are added as special (leaf nodes). The example of a CCG is shown in Figure 3.2.

However, when a function is being called from within a loop it will result in as many sub-graphs as there are loop iterations. Usually these sub-graphs are almost identical, which

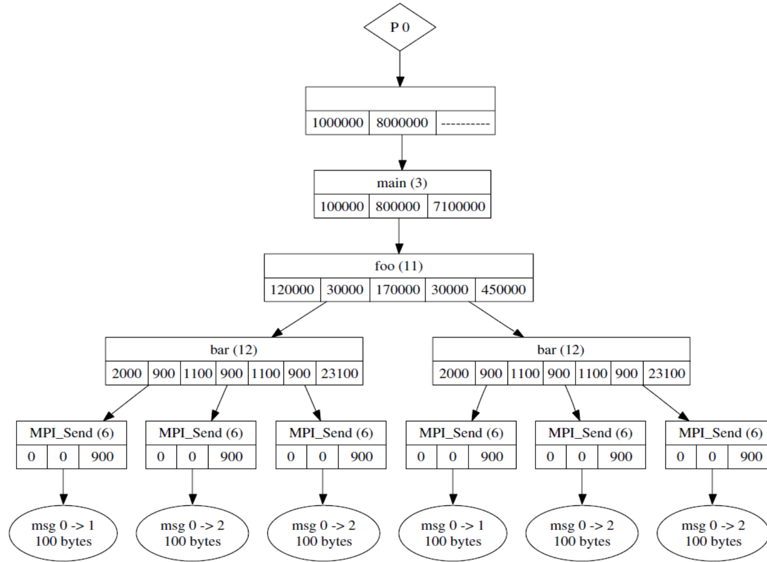


Figure 3.2: CCG of an example function[37]. For the compressed version see Figure 3.3

allows to represent all similar sub-graphs by only one instance. The process of searching similar sub-graphs is done in parallel. Furthermore, the comparison operator, producing the similarity measure is implemented in a very efficient way such that it has a constant computational complexity regardless of the number of comparisons. The similarity function performs the comparison of the sub-graphs using *hard* and *soft properties*. The hard properties are the semantic properties such as a function id and have to be identical in order to merge two sub-graphs together. The soft properties, such as execution time, are similar when the difference is within a certain error-range. The requirement of exact match of hard properties allows to use hashing, which enables the constant comparison complexity mentioned above. The error of matching the soft properties is propagated up the hierarchy and accounted in the parent node. This allows to ensure that the over-all error for any sub-graph will not exceed the allowed limit. The compressed version of the example given in Figure 3.2 is shown in Figure 3.3.

The experiments on the real-world application traces showed compression ratios (in terms of memory reduction) from 2.7 with an error tolerance interval of 0, i.e. no information loss, up to 262 in case of a lossy compression.

Furthermore the cCCG features a straightforward querying mechanism. Since the structure of the cCCG is similar to the original CCG, no decompression is needed. This makes cCCG an important infrastructure enabling efficient interactive trace exploration inside VampirNG.

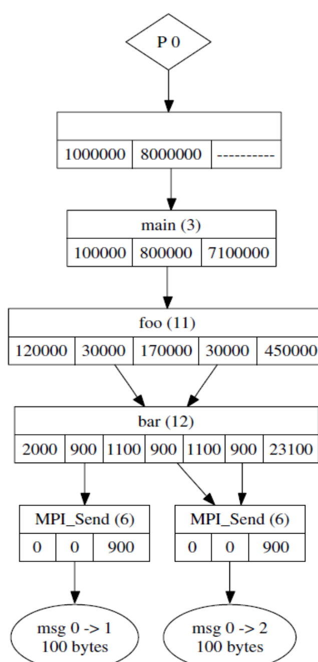


Figure 3.3: Compressed version of the example function shown in Figure 3.2 [37].

3.2 Analysis

The compression techniques presented above allow to mitigate the issue of data size growing with time. This makes possible recording and storing performance measurements of longer application runs. However, it doesn't solve the bottleneck associated with the temporal dimension, but rather shifts it to the next stage, namely the analysis process. Now, the collected and compressed performance measurements have to be evaluated. Often this tedious task is left to the user. In this section we present a number of techniques assisting the user in overcoming this challenge.

3.2.1 Detection and Application Structure Extraction

One of the approaches assisting the user in analysis of long traces is the automatic detection of the trace structure. The value of the resulting information is two-fold. First, it allows an easier comprehension of the traces and a faster identification of intervals of interest. Second, in case of a repetitive behavior, the technique, instead of presenting multiple repetitions of the same pattern, presents only one - the most representative instance of the pattern.

Structure Extraction Using Signal Processing

In [21] a technique based on signal processing algorithms is presented. The technique automatically extracts the internal structure of an application by analyzing periodical and hierarchical properties of performance signals generated from a trace.

In the first step the full trace of an application is collected. This is then used to generate a set of relevant performance signals which characterize the temporal evolution of a given metric. Authors find the following metrics useful in detecting the application structure: instantaneous number of application processes performing computation, point-to-point communications or collective communications; instructions per cycle, instantaneous number of outstanding messages.

In the next step the computation phase of the application is detected and separated from the initialization and the finalization ones. The differentiation is made based on the assumption that the computational phase contains higher frequencies in the above metrics. Since the temporal location of high frequencies is of interest, in order to detect the beginning and the end of the computational phase, Wavelet analysis was used (for more details on the Wavelet analysis see Section 6.2). Actual detection of the computational phase is done by selecting time intervals based on the wavelet coefficients characterizing high frequencies. The selection of the intervals is parameterized using two parameters λ and δ. The first parameter defines a threshold for selecting significant coefficients. For example, the value $\lambda = 0.3$ means that all the coefficients which are greater or equal to 30% of the maximum value are selected. The value of the δ parameter defines the width of the time interval around the selected coefficient to be marked as a part of the computational phase. Therefore, the union of all the intervals gives the temporal domain of the computational phase.

After the temporal location of the computational phase is identified the performance signals are further analyzed in order to identify repetitive patterns. This is achieved by means of the autocorrelation function $A(k)$ [60]. The function reaches its local maximums near the values k equal to the main periods observed in the signals. By selecting maximums of the autocorrelation function one can detect periodicity of the present repetitive patterns. In order to make sure that the selected maximums correspond to a meaningful pattern, authors apply two heuristics. The first one requires that the detected maximum is at least 10% higher than any other local maximum. The second checks the value of the autocorrelation function for the second harmonica which is a multiple of the detected period. If the period $2T$ is also a maximum, then the period T is accepted as the period of the detected iterative pattern. The analysis is then performed recursively within the single iterations of the detected pattern searching for the nested repetitive patterns.

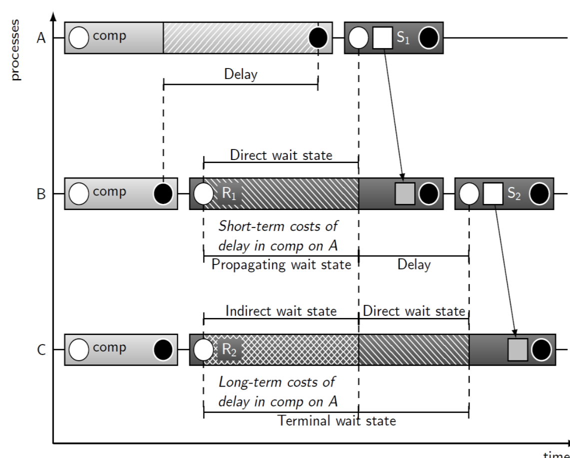


Figure 3.4: Time line diagram showing simplified event sequence on three processes. Shaded rectangles indicate certain activities defined by the corresponding events. These are circles for region enter and exits and squares for send and receive events. The arrows indicate the direction of the messages. It can be observed that extended execution time of the "comp" activity on process A leads to direct waiting time in the process B. This waiting time plus an additional delay caused by the receive operation on process B is then responsible for the waiting time in the process C [19].

3.2.2 Root Cause Analysis

Another approach to the automatic analysis of temporal performance dynamics originating from the propagation of MPI wait states over the time and process dimensions is taken in Scalasca [19], [42]. The technique scalably identifies the original cause of the observed wait states as well as quantifies induced direct and indirect costs.

Figure 3.4 shows a fragment of the application trace highlighting a wait state situation and how it evolves over time and processes. Here one can see that an extended execution time of the "comp" activity on process A leads to a delayed send operation which in its turn is responsible for the waiting time in the corresponding receive on process B. The wasting of time in waiting is, however, not limited to process B. The delay in the completion of the receive on process B delays also the send operation to the process C, which is already delaying the corresponding receive. Here one can see that part of the waiting time suffered in the receive on process C is indirectly caused by the original delay in the completion of the "comp" activity on process A. This example nicely demonstrates how wasted time in waiting is evolving over the time and migrating from one process to another.

Based on the cause and the consequences, authors formalize the three following types of a wait state:

- *Direct wait state* - a wait state caused by any activity which is not a wait state itself
- *Indirect wait state* - a wait state caused by another wait state
- *Terminal wait state* - a wait state which doesn't cause other wait states

The activity or a part of an activity causing the process to be late at the synchronization point and therefore causing a direct wait state on the peer process or set of processes is called a *delay.* Using the terms presented above authors define a *causation chain* as a sequence of a delay, a direct wait time, indirect wait times and a terminal wait time.

In order to quantify the wasted resources due to a delay along the causation chain a notion of *costs* is introduced. It is further refined into the *short-term costs* covering the time wasted in direct wait states and the *long term costs* which are the sum of all indirect wait states.

Therefore, the goal of the root cause analysis is to map the costs of a delay onto the call-path and the processes where the delay occurs. This is achieved by the following algorithms:

1. Collection of a trace
2. Parallel forward trace replay
3. Parallel backward trace replay

During the trace collection the execution of the application is monitored and the relevant events, including the communication events, are stored into a trace file. During the second step the trace file is read by a parallel analyzer and it is "replayed", i.e. the communication events stored in the trace are consequently repeated, but instead of the original payload the time stamps of the respective communication functions are communicated. This allows to detect wait states by comparing the time-stamps of the peers. In the third step, the trace is replayed again, but now in the backward direction. Here the wait states identified in the previous step are classified according to the aforementioned types and linked together into causation chains. The beginnings of the causation chains point to the delays, i.e. the root cause.

Root cause analysis is a powerful technique directly pointing to the location of the inefficiency and not the symptoms of it, which are wait states in this case. Additionally, it quantifies the associated costs. However, authors recognize that the application of the technique is not scalable along the time dimension [19] due to the limitations of the tracing approach. The solution they suggest is to limit the tracing, and therefore the root cause analysis, to particular time intervals where the wait states manifest themselves. In particular they suggest to use dynamic profiling in order to identify these intervals.

3.3 Summary

The techniques described above target the challenges arising when the temporal resolution in performance analysis is of interest. The most critical issue is the size of performance data linearly growing with time. Therefore, many techniques were developed to cope with it by means of data compression along the time dimension. The compression techniques like cCCG and Dynamic Profile Clustering utilize repetitiveness in performance measurements to shrink the data size both exact and lossy. Alternatively, the compression algorithm for load-imbalance measurements described in Section 3.1.2 relies on proven signal processing algorithms such as Wavelet analysis. Described techniques demonstrate significant compression rates, in some cases even time-independent constant size of the compressed datasets is achieved.

However, the compressed raw performance data has to be analyzed in order to extract the knowledge about performance inefficiencies and optimization potentials. The bottleneck of the time-dependent growth in data size is now shifted to this step. We overviewed two different techniques simplifying the analysis of temporal performance data. The first one automatically detects regularities in the trace data. This simplifies comprehension of the measurements. Additionally it allows to limit the analysis time window to an interval which is a representative of a repetitive pattern. The second technique takes another approach. It focuses on the specific problem of MPI wait state analysis. This can dynamically degrade over time and migrate to other processes. The technique automatically parses the trace in order to first identify the wait states and then track them back to the originating cause which is responsible for the problem. Moreover the direct and indirect costs in terms of the time wasted in wait states are quantified.

The information about how does the performance change with time, however, cannot be easily available from the techniques described above. The evolution of characteristics is particularly difficult to extract manually since it requires analysis of the time-dependent data. On the other hand, this information is valuable for performance tuning, since it allows to answer the following questions:

- Is a particular performance characteristic dynamic?
- When and where are the performance degradations suffered?
- How do they impact the overall performance?

In the next chapters we present a number of techniques and their implementation capable of providing answers to the questions above automatically.

Chapter 4

Automatic Instrumentation Adaptation

4.1 Introduction

Performance monitoring, as any other measurement process, relies on the insertion of probes which distort the original behavior of the observed system. In this case the inserted probe functions are executed by the same core and by that affect the application behavior. This negative influence, called overhead, affect different aspects of the execution. Time is one of the most impacted characteristic which can grow by orders of magnitude when too much instrumentation is introduced. However the aspects like memory behavior, parallel coordination between processes and branch prediction are being corrupted as well. The automatic overhead elimination technique proposed in this work is targeting overheads which manifest themselves in increased execution time due to measurement intrusions. It could be also extended to address other aspects of performance measurement overheads.

The techniques presented in this chapter improve the method of direct instrumentation discussed in Section 2.3.2. In particular, source level direct instrumentation was the main focus for the development of the novel automatic instrumentation adaptation approach. Until now overhead mitigation and instrumentation adjustment was the task carried out manually by the user. In case of Score-P (a joint instrumentation and measurement infrastructure for Scalasca, Vampir, Periscope and TAU) user can filter out high-overhead regions by manually inserting high-overhead instrumentation regions to the black-list of a filter file. In case of the commodity instrumentation and measurement infrastructure of Periscope, called MRIMonitor, the only possibility to control inserted instrumentation was to allow instrumentation of particular region types, e.g. loops, subroutines, calls, for each source file separately. Although, the mechanisms proved to be efficient in many cases, the manual approach requires significant efforts by the user.

Another significant disadvantage of the existing techniques for overhead reduction, are

that they don't consider the type of the measurements and analysis being performed. For example, the artificial wait-states created by an un-even amount of overhead on two tightly communicating processes are of course a severe problem when MPI performance is of interest. However, one can live with the negative effect when the single-core or the node-level performance is a subject of the performance analysis. Moreover, tools like Periscope collect measurements at different instrumentation locations at different stages of the on-line analysis process. This makes instrumentation of other locations needless. The novelty of the proposed automatic instrumentation adaptation strategies presented in this work lies in the ability to consider the specifics and the run-time needs of the performance analysis executed by the tool.

In order to demonstrate our techniques we implemented them in the framework of the Periscope instrumentation and measurement infrastructure MRIMonitor. In comparison to Score-P, it allows selective recording, i.e. measurements can be configured to be collected on a specified sub-set of instrumented regions. Also the implementation of the probe functions, allowing cancellation of a part of the introduced overhead, makes overhead estimation more difficult in case of the MRIMonitor. Finally, the multi-step online analysis process performed by Periscope together with the automatic restarting of applications allows a tight integration of the proposed instrumentation adaptation strategies and the analysis ones.

4.2 Overhead Model

In the first step we develop a model of overheads introduced by monitoring an application. Though, the model was developed for the monitoring infrastructure of Periscope, it can be easily adapted to other tools.

Following the direct instrumentation approach calls to the probe functions `start_region` and `end_region` are inserted before and after the region. Typical activities performed by a monitoring library within the `start_region` are shown in Figure 4.1. The monitor function `end_region` has a similar structure.

First the Periscope monitoring library checks whether there are any measurement requests (1) for the region to be entered. If the analysis agent did not request any measurements, there is no need to enter the measurement management. Second, it checks whether the region is the phase region. If so, the application might be suspended for on-line measurement configuration. If both conditions are false, the application can directly continue. The overhead for these two checks is called the *prolog overhead*.

In the next, measurement counters are stopped (3) and read (4), in order to be configured according to the requests specified for the region to be entered (5,6), the necessary monitoring bookkeeping is performed (7) and finally the counters are started again (8). This part of the overheads is denoted as *request overhead*.

```
Function start_region(Region reg)
{
  prol_ovhd {  1.  Check whether there are request for region
               2.  if (no request and no control point) return
  req_ovhd  {  3.  Stop counters
               4.  Read counter values
               5.  For all request
                  6.  Configure counters
               7.  Perform measurements bookkeeping
               8.  Start counters
}
```

Figure 4.1: Pseudo code of the monitoring functions and the overheads.

By stopping and starting the counters the major part of the *request overhead* is canceled out from the local measurements. Nevertheless, this time is still amortized in the extended execution time of the monitored application and potentially in other processes due to delayed synchronization. The overhead for the request management depends very much on the number of request and the request type.

The composition and comparison of the pure execution time, observed and measured execution times as well as the prolog and request overheads of an arbitrary region nest are shown schematically in Figure 4.2. Here the execution time, in abstract time units, goes from top to bottom. In the opposite direction cumulative values for the metrics above are plotted with hatched rectangles.

The estimation of the introduced overheads is based on the number of calls to instrumentation functions. These are easy to obtain and associated overheads are very low. For an arbitrary region nest the following two metrics are needed to estimate introduced overheads:

- *Exclusive instances* (`excl_instances`) - the number of instances when the overhead was introduced due to the probe functions before and after the given region.
- *Nested instances* (`nested_instances`) - the sum of `excl_instances` of all nested overhead occurrences during the execution of the given region.

In order to obtain the time spent due to each type of overhead, Periscope is calibrated at the configuration time. In this procedure the time associated with a single instance of the prolog overhead (`prol_ovhd`) and the request overhead (`req_ovhd`) are measured.

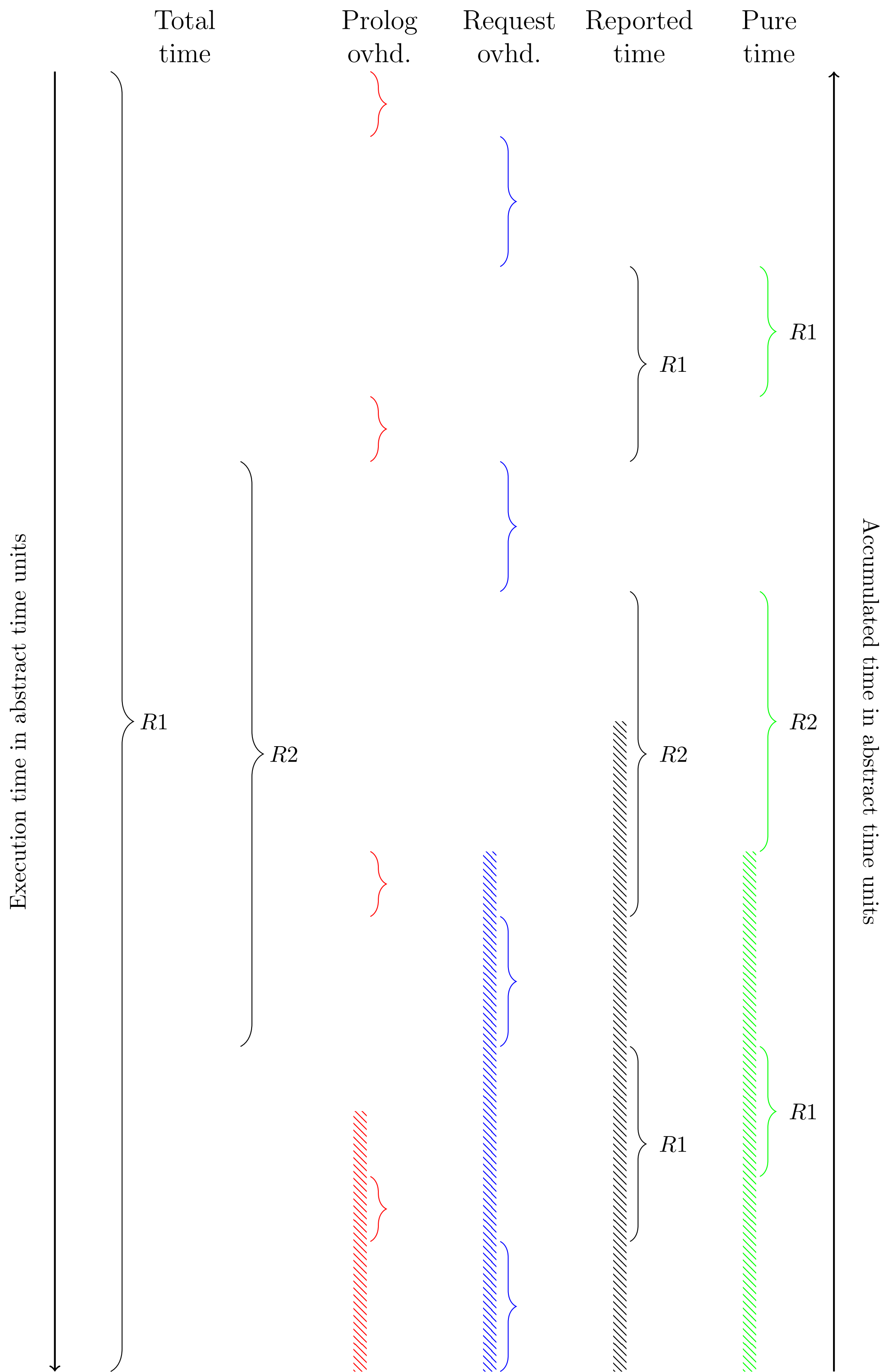


Figure 4.2: Break down of the total, reported and pure times of a region nest (labeled as R1 and R2) together with the generated prolog and request overheads.

4.3 Instrumentation Strategies

Based on the overheads model presented above three strategies for automatic minimization of overheads were designed and implemented in Periscope:

- Total Overhead Reduction (TOR) Strategy: limits the increase of the observed execution time
- Prolog Overhead Reduction (POR) Strategy: limits the increase in measured time due to the prolog overhead, i.e., the non-corrected part of the measurements.
- Analysis Guided Overhead Reduction (AGOR) Strategy: adjusts instrumentation according to the on-line needs of the analysis strategy.

In the next sections we describe each strategy in details.

4.3.1 Total Overhead Reduction Strategy

Figure 4.3 illustrates the algorithm of the TOR Strategy. This strategy is based on the monitor's own overhead estimations and limits the increase of the execution time, as it can be measured on the application level.

First, all the regions of interest are instrumented. This is controlled by the programmer via the configuration file. This instrumented version is then started by the frontend. The frontend instructs the analysis agents to search for regions with excessive overhead. Thus, the agents request appropriate measurements and the application's phase is executed. The measurements are then retrieved and the agents compute overhead estimations according to the model above. Then the *Excessive Total Relative Overhead* property is evaluated against the overhead estimates for all instrumented regions and is accepted when the total overhead relative to the pure execution time of a region is greater than a certain threshold. The specification of the property is provided in Figure 4.4.

In order to evaluate the property, the pure execution time of a region is computed. It is computed from the measured execution time of the region (`exec_time`). While measuring it the request overhead was already canceled out, however, it still includes the prolog overhead, which has to be accounted for. First we subtract the prolog overhead of the probe function `end_region` behind the region itself. This overhead is included in the execution time since the counters are only stopped after the prolog of this probe function. The prolog overhead of the `start_region` is not included in `exec_time` as one can see in Figure 4.2. Thus, we multiply the estimate of the prolog overhead (`prol_ovhd`) with the exclusive instrumentation instances of the region (`excl_instances`). We also have to subtract the prolog overhead of all dynamically nested monitor library calls. Since in this case both `region_start` and `region_end` polluted the measurements with their prolog

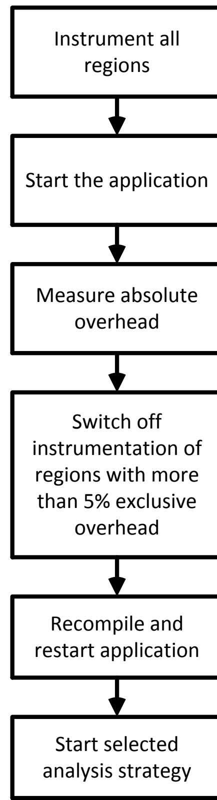


Figure 4.3: Control flow of the TOR Strategy.

overheads we have to account two `prol_ovhd` per instance of a nested instrumentation. Using the estimation of the region's own total overhead and the computed `pure_time` we can compute the relative overhead of the measurements for this region.

After the *Excessive Total Relative Overhead* property is evaluated for all the regions, the instrumentation is removed for those regions where the property was found. Then the application is recompiled and the standard analysis process of Periscope is continued.

The TOR strategy limits the increase in execution time of the application due to overheads of the direct instrumentation. It can be used in combination with any search strategy, but, in particular, it is highly beneficial in case of MPI or OpenMP analyses. The explanation is that it reduces the artificial load imbalance introduced by an un-even growth in the application execution time due to overheads on different processes or threads. The effectiveness of the strategy is easy to check via direct wall clock time measurements of the application. The drawback of the strategy is that it might remove instrumentation with high total overhead but still allowing valid measurements achieved by the built-in cancellation of the request overheads (see Section 4.4 for an example). The next strategy provides a compromise solution for this problem.

```
PROPERTY Total_overhead(int exec_time, int excl_instances, int
nested_instances)
{
 LET
     pure_time = exec_time - excl_instances * prol_ovhd -
     2 * nested_instances * prol_ovhd;
     rel_ovhd = excl_instances * 2 * (prol_ovhd + req_ovhd) * 100 /
pure_time;
 IN
  condition  : rel_ovhd > threshold
  confidence : 1.0;
  severity   : rel_ovhd;
}
```

Figure 4.4: Specification of the “Excessive Total Relative Overhead“ property

```
PROPERTY Prolog_overhead(int exec_time, int excl_instances, int
nested_instances)
{
 LET
     pure_time = exec_time - excl_instances * prol_ovhd -
     2 * nested_instances * prol_ovhd;
     rel_ovhd = excl_instances * 2 * prol_ovhd * 100 / pure_time;
 IN
  condition  : rel_ovhd > threshold
  confidence : 1.0;
  severity   : rel_ovhd;
}
```

Figure 4.5: Specification of the “Excessive Prolog Overhead“ property

4.3.2 Prolog Overhead Reduction Strategy

The Prolog Overhead Reduction strategy, for short POR strategy, limits the impact of overheads on the measurements produced by the monitoring library. It follows the same steps as the TOR strategy described above, but instead of removing instrumentation with high total overhead, it eliminates overhead sources responsible for severe alternation of measurements due to the non-corrected overhead part, i.e., the prolog overhead. Following the property-based analysis approach of Periscope, the property *Excessive Prolog Overhead* was created to automatically detect the regions where instrumentation is responsible for high prolog overheads. The specification of the property is presented in Figure 4.5.

Similar to the Excessive Total Relative Overhead property described in Section 4.3.1, the pure execution time (`pure_time`) of a region is computed first using the same formula.

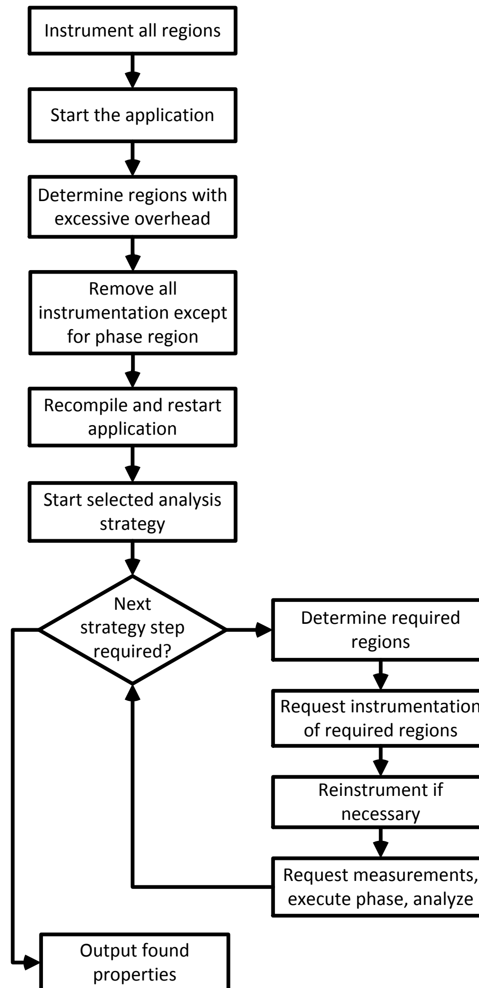


Figure 4.6: Control flow of the AGOR strategy.

The relative overhead (`rel_ovhd`) is , however, computed accounting only for the prolog part of the overhead generated by the region instrumentation. In this case the `rel_ovhd` would be much lower than the value of the relative total overhead computed for the same region. This results in less instrumentation being removed by the strategy.

POR strategy allows more fine-grained measurements, but at the same time it ensures that the overheads introduced to process-local measurements are within acceptable limits. This makes it a perfect instrumentation adaptation solution in case of a single core performance analysis.

4.3.3 Analysis Guided Overhead Reduction Strategy

The AGOR instrumentation strategy implements another alternative approach to the overhead reduction. As it follows from the name, the overheads are reduced by means of adapting instrumentation to the ongoing analysis process of Periscope guided by a given analysis strategy. The control flow of the AGOR strategy is shown in Figure 4.6.

First, all regions of the application are instrumented. Then either the TOR or the POR strategy is used to filter out the regions which instrumentation generates unacceptable overheads. These regions are then added to the black list and are never instrumented.

After the high-overhead regions are identified, instrumentation of all regions except the root region, called the phase region, is removed and the control is given to the requested analysis strategy.

Typically the analysis strategy is a multistep strategy. In each step it automatically determines the regions which need to be evaluated in the current step. These regions are then transparently communicated to the instrumentation strategy which checks whether the regions are in the black list, and if not, their instrumentation is enabled. Whenever a new region needs to be instrumented, the instrumentation of the regions which are not anymore needed by the analysis strategy is removed. In the Periscope implementation, every time the new instrumentation is added or removed, the application is recompiled and restarted. After that, the analysis strategy collects the necessary measurements and evaluates current hypotheses against them. Based on the results of this evaluation another round of analysis on a new set of regions might be requested by the analysis. In this case the procedure of instrumentation adaptation is repeated.

AGOR strategy features the lowest possible overhead among all the known direct instrumentation techniques. It is achieved by, first, removing all instrumentation which is not of interest at the current state of the analysis. The instrumentation in this case is entirely removed, which is different to the approach taken in ParaDyn. Second, only the instrumentation of the regions which are currently of the analysis interest (typically a very small set) are instrumented and only in the case when the associated overheads are within a certain threshold. The obvious drawback of this strategy is that the entire time for the analysis might dramatically increase due to multiple recompilations.

4.4 Results

We demonstrate the effectiveness of the proposed automatic adaptation of instrumentation strategies using two applications. The first one is a synthetic benchmark simple enough to "look under the hood" and to demonstrate step by step our approach. Then with the second application - a large particle simulation code - we show the benefits of our technique in a real-world setup.

We run our experiments on the Altix 4700 supercomputer which was operational at Leibniz Supercomputing Centre (LRZ) in Garching from 2007 till 2012. Although, at the time this thesis is written the system is not anymore in service, the results presented below can be easily extrapolated to current machines, since the performance and the actual characteristics of the system used for the experiments are irrelevant to the techniques presented here.

We combined the instrumentation strategies presented above with two analysis strategies: the multi-step *Stall Cycle Analysis* (SCA) strategy and the single step *Stall Cycle Analysis Breadth First* (SCABF) strategy.

The multistep SCA strategy is based on the Itanium's stall cycle counters. They count the number of stall cycles in the processor's pipeline due to different reasons and are organized in a hierarchy. For example, the `BACK_END_BUBBLE_ALL` counter determines the number of all lost processor cycles. On the next level, special counters determine stall cycles due to special events. For example, `BE_EXE_BUBBLE.GRALL` determines the stall cycles for waiting for the delivery of data to a general purpose register. `BE_EXE_BUBBLE.FRALL` determines the stall cycles for waiting for the delivery of data to a floating point register.

The two search strategies basically follow the incremental search of refining performance properties down this hierarchy. The SCA strategy starts for the program's phase region. It first checks whether there is a significant number of stall cycles. If not, the search terminates. Otherwise, the search refines in the property hierarchy, looking for stall cycles due to memory access overhead, exceptions, TLB misses, and so on. Thus, counters are not wasted for checking unnecessary performance properties.

The second strategy SCABF generates in the first search step all stall cycle-related properties for all regions of the program. Due to the 12 Itanium counters, all can be measured in a single execution of the phase.

4.4.1 Nested Loop Example

The first example is a little program shown in Figure 4.7. It has an iterative phase (line 22) which is marked by the directives `USER REGION` and `END USER REGION` indicating the repetitive analysis phase of Periscope. This phase region is executed 30 times. It consists of a single loop in line 24 which has two nested loops in line 27 and line 31. The second loop is more compute intensive since it has 30.000 iterations while the first only modifies the first 1000 elements. The execution time of the phase is externally measured via the inserted wall clock time functions of MPI. Thus, we can compare the information measured by the monitor with the reference measurement in the phase region.

First, we let Periscope instrument all the regions in the test code. In this case the regions starting at lines 22, 24, 27 an 31 are automatically instrumented by inserting calls to the probe functions `region_start` and `region_end` immediately before and after the region. We evaluate only the TOR and the POR instrumentation strategies, since the code has

```
20: do k=1,30

22: !$MON USER REGION
      ts=MPI_WTime()
24:   do l=1,1000
        s=0.0

27:     do i=1,1000
          a(i)=500*i
        enddo

31:     do i=1,30000
          s=s+a(i)
        enddo
      enddo
      te=MPI_Wtime();
      write(*,*) te-ts
    !$MON END USER REGION
    enddo
```

Figure 4.7: Example code with three nested loops.

only three regions inside the user region and, therefore, is not suitable for the multi-step refinement supported by the AGOR strategy.

Table 4.1 shows the execution time of the phase region (line 22) measured via MPI_Wtime as well as the corresponding value measured by the monitoring library. The presented values are obtained after the instrumentation was adapted by the TOR and POR strategies respectively.

Here we can see that in the first case the reference execution time is equal to 0.17 seconds which is equal to the execution time of the uninstrumented application. In case of the POR Strategy the reference measurements show much higher values indicating severe growth in the execution time due to overheads. Nevertheless, the values reported by the

Table 4.1: Phase region execution time during overhead estimation and the subsequent analysis step

Instrumentation Strategy	Reference time, sec.	Reported time, sec.
No instrumentation	0.17	–
POR Strategy	0.35	0.18
TOR Strategy	0.17	0.17

Table 4.2: Relative total and prolog overheads of the code regions in % of the pure time.

Strategy	Total overhead, % of the phase time	Prolog overhead, % of the phase time
User region at line 22	0.3	0.003
Loop at line 24	0.3	0.003
Loop at line 27	1085	18.4
Loop at line 31	100	2.7

Table 4.3: Properties found by the SCA analysis strategy supported by the TOR and POR instrumentation adaptation strategies.

Region	Property	Severity	
		TOR Strategy	POR Strategy
User region at line 22	IA64 Pipeline Stalls	28.5	27.8
	Stalls due to L1D TLB misses	23.1	23.0
Loop at line 24	IA64 Pipeline Stalls	27.7	27.8
	Stalls due to L1D TLB misses	23.1	23.0
Loop at line 27	–	–	–
Loop at line 31	IA64 Pipeline Stalls	–	24.9
	Stalls due to L1D TLB misses		21.2

monitoring library are valid.

The observations above can be explained by the estimated overheads presented in Table 4.2. Here we can see that the overheads of the regions at lines 27 and 31 are very high compared to the respective execution times. As discussed above it is responsible for the growth in the execution time but at the same time it is almost completely canceled out from measurements collected by the monitoring library.

In case of the TOR strategy the instrumentation is removed from both regions. This explains why we observe almost no overhead, neither in the reference measurements nor in the monitor ones.

In case of the POR Strategy, the relative prolog overhead, as it can be expected, is not that severe. Since it is above the threshold of 5% for the loop at line 27 the instrumentation of this region is removed. For the loop at line 31 it is only 2.7%, which is below the threshold and, thus, the instrumentation is preserved. However, the loop at line 31 still has a total overhead of 100% which explains the severe growth in the reference time. The values delivered by the monitor are, nevertheless, correct.

Table 4.3 compares the performance properties found. The second column identifies the property either *IA64 Pipeline Stalls* identifying a situation where many stall cycles occurred and *Stalls due to L1D TLB misses* where TLB misses induce stall cycles. The next two columns report the severities of these properties for analysis runs combined with

Table 4.4: Phase time and the number of high-overhead regions removed for the analysis runs with no instrumentation strategy, TOR strategy, POR strategy and a reference run without any instrumentation.

Instrumentation strategy	Phase time, sec.	#regions removed
None	150	n/a
TOR Strategy	3.26	110
POR Strategy	3.47	62
Uninstrumented	3.18	n/a

the TOR and POR strategies. We see that both analysis runs report the same properties with almost the same severity for the user region and the loop in line 24. For Loop 27 none of the runs report any properties. Obviously, the reason is that in both cases the instrumentation was removed. Nevertheless, by comparing the severities of properties found for the user region and the loop at line 31 (which is together with the loop at line 27 nested) we can conclude that the contribution of the loop at line 27 is insignificant. For Loop 31 only the second analysis run reports properties since the instrumentation was removed by the TOR strategy in the first run.

The results presented in this section demonstrate the difference between the TOR and the POR instrumentation strategies. The first one allows efficient elimination of the overhead manifesting itself both in the execution time growth as well as in the analysis results produced by Periscope. On the other hand, the POR strategy, still ensuring correct analysis results, allows more fine grained analysis by preserving more instrumentation.

4.4.2 PEPC

PEPC [35] is one code in the DEISA Benchmark Suite [2]. It is a parallel tree-code for rapid computation of long-range (1/r) Coulomb forces for large ensembles of charged particles. The heart of the code is a Barnes-Hut style algorithm employing multipole expansions to accelerate the potential and force sums, leading to a computational effort O(NlogN) instead of the O(N^2) which would be incurred by direct summation. Parallelism is achieved via a 'Hashed Oct Tree' scheme, which uses a space-filling curve to map the particle coordinates onto processors. The version we used in our experiments is PEPC-E which consists of 66 FORTRAN 90 files (13.000 LOC) plus a library for tree management.

The first column of the Table 4.4 presents the execution times of the phase region of PEPC-E for the analysis runs with and without instrumentation strategies as well as a reference run with no instrumentation at all. Here we can see that when no instrumentation strategy is used, the execution time of the phase time grows by a factor of 45. By applying the instrumentation strategies the extreme growth in the execution time is cured and becomes almost equal to the reference run with no instrumentation. In the second column we can

Table 4.5: Properties and their severities found by the SCABF analysis strategy in PEPC with and without instrumentation strategies.

Region	Property	Severity		
		TOR	POR	None
pepce.f90:82	IA64 Pipeline Stalls	60.2	58.2	50.7
	Stalls due to L1D TLB misses	37.8	32.4	–
fields.f90:15	IA64 Pipeline Stalls	60.1	57.7	50.7
	Stalls due to L1D TLB misses	37.7	32.4	–
fields.f90:209	IA64 Pipeline Stalls	52.9	53.1	42.9
	Stalls due to L1D TLB misses	32.6	30.1	–

see the number of instrumentation regions removed by the TOR and POR strategies. Although, the TOR Strategy removes almost twice the amount of regions removed by the POR Strategy, the difference in the overhead reduction is not as dramatic as in the previous example. This can be explained by the fact that most of the regions removed by the TOR Strategy and kept by the second one are slightly above the threshold and do not generate significant overheads. Nevertheless, when compared to the analysis run with no instrumentation strategy, the phase time drops from 150 to 3.26 seconds in case of the TOR strategy and 3.47 seconds in case of the POR Strategy.

Table 4.5 presents the performance problems found in PEPC on process 0. The results of the other processes were similar. The analysis algorithm used was SCABF which checks all the properties for all regions in one phase execution. The threshold for a property to become a performance problem is a severity value of 30% of the phase time. As we can see from the table, the reported severities for both runs with an instrumentation strategy are quite similar and significantly higher than those reported without using an instrumentation strategy. We can also see that the analysis with an instrumentation strategy reports one more specific performance problem for the three first regions. The reason is that the prolog overhead influencing the measurements leads to the underestimation of the severity of evaluated properties and thus to false-negatives, i.e. the situations when some relevant properties are not reported.

4.5 Summary

Measurement overheads is an important concern for performance analysis tools. When too many intrusions perturb observed application execution, obtained results may be corrupted. Particularly, direct source code instrumentation techniques are vulnerable in this respect since no simple mechanism for controlling overheads is available. The existing techniques based on selective instrumentation or runtime filtering of high-overhead regions require manual input from the user and often require multiple try and error iterations.

In this chapter we presented an automatic instrumentation adaptation strategies for overhead reduction. The novelty of the presented approach is two-fold.

First, it presents a model and a lightweight scheme for overhead estimation. Here the overheads introduced by the monitoring system are broken down into two components: prolog overhead and request overhead. The first one comprises the overheads accumulated in the observed performance measurements due to executing a probe function. The second component represents the overheads introduced by reconfiguring measurement counters and performing intensive bookkeeping typical to on-line analysis tools. This part of the overhead is often much bigger than the prolog one, but, when properly implemented, is canceled out from the measurements. In order to compute the estimates we propose to extend the monitoring library to provide lightweight measurements of the probe function calls exclusively for each instrumentation region as well as the number of nested instrumentation calls.

Second, we develop three automatic instrumentation adaption algorithms, called instrumentation strategies. The novelty of our approach is that the decision on automatic removal of the instrumentation is based not only on overhead estimations but also on the type and the runtime needs of the employed performance analysis process. This allows to mitigate exactly the impacts of the introduced overhead which are critical for the current analysis.

For example, when the MPI wait-state analysis is of interest it is important to minimize the un-even extensions of the application execution time due to introduced overheads. This is achieved by the Total Overhead Reduction strategy which removes instrumentation regions when total overhead, comprising both prolog and the request one, is above a certain threshold. In contrast, when single core performance is of interest, the precision of the obtained measurements and the granularity of the instrumentation are important. Prolog Overhead Reduction strategy achieves this by only removing regions which generate high prolog overheads, which are typically only a small subset of the regions removed by the previous strategy. Finally, when multi-step on-line performance analysis is performed, the instrumentation is needed only around the regions which are currently in the focus. For such scenarios the Analysis Guided Overhead Reduction strategy was developed.

The concepts presented in this chapter were implemented in Periscope performance analysis tool and evaluated with a real-world application. Our experiments show that by automatic reduction of overheads with the presented instrumentation adaptation strategies we achieve more precise performance analysis results. First, the severity of the performance problems found in experiments with instrumentation strategies are corrected to higher values when instrumentation strategy is applied in comparison to experiments when no strategies were used. Second, in experiments with instrumentation strategies Periscope detects additional performance problems which are blurred by overheads otherwise.

Chapter 5

Temporal Scalability of Performance Dynamics Analysis

The size of temporal performance data growing linearly with time poses a severe challenge specially in case of long running applications. The technical difficulty of collecting it as well as associated overheads are prohibitive factors making performance dynamics a hard aspect to evaluate.

5.1 Introduction

Various schemes were developed to handle the issue. The profiling technique solves the problem by simply shrinking the time dimension by taking various statistics over the time. This solves the problem of the linearly growing size at the cost of compromising the time resolution and could be seen as one extreme. The other extreme would be tracing where every single event is recorded and stored explicitly in a time-stamped buffer. This gives the best possible resolution but suffers most from the poor temporal scalability.

The solutions lying in between of the two extremes were presented in Chapter 3. Although, relying on different principles and properties of the application, the common approach is based on the on-line compression of the temporal data by exploiting the repetitive behavior of the application which is typical for numerical simulations. The result is a trade-off between the size of the data and the obtained temporal resolution at the cost of additional computations required for the on-line processing.

In this chapter we present a new scheme for the analysis of temporal performance data called *Online Dynamic Profile Analysis* (ODPA) allowing to overcome the issues mentioned above. The technique features time-independent constant data size; temporal resolution of dynamic phase profiling; and zero processing overheads on the application side due to remote on-line processing. The benefits come at the cost of additional resources

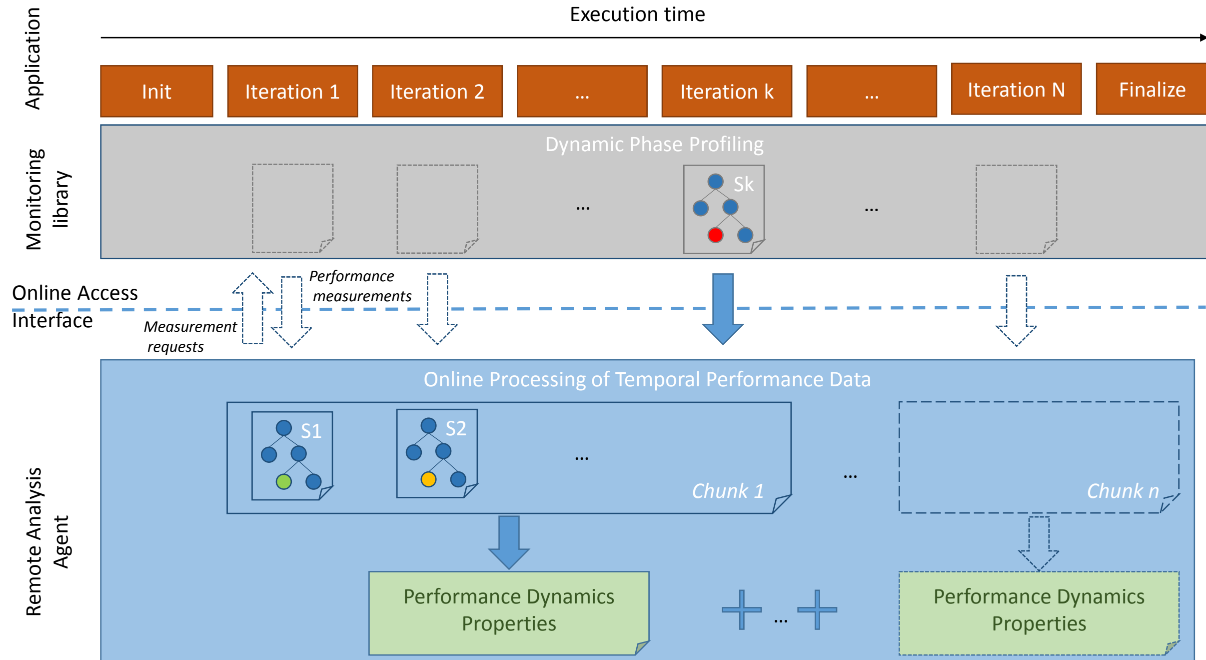


Figure 5.1: Online dynamic profile collection and analysis scheme design.

for remote analysis units.

5.2 Design Overview

A high-level overview of the ODPA scheme is presented in Figure 5.1. In our approach we separate the processes of temporal performance data generation, performed by a monitoring library, and its analysis which is happening on the fly on a *Remote Analysis Agent* (RAA). This is different to other approaches to the performance dynamics analysis where the data is compressed online in order to be postmortem analyzed, either manually or automatically.

On the monitoring side of the scheme we utilize the repetitive nature of applications to obtain samples of temporal performance measurements. Here we extend the *Dynamic Phase Profiling* technique which is used to collect a separate profile sample for each iteration of the progress loop of the application. At the end of the iteration a profile sample is transferred over an *Online Access Interface* (OAI) to the (RAA) and immediately discarded on the application side. Thus, the amount of data stored on the application side is decoupled from the number of profiled iterations.

By sending samples of the dynamic profile to the RAA the bottleneck, however, is not completely removed but only shifted to another location. In order to completely resolve it, the temporal performance data has to be also processed in online. We achieve this by accumulating a predefined amount of samples of dynamic profiles and then processing them in chunks. During the processing the chunks are searched for relevant performance dynamics characteristics and patterns. These are then preserved as *Performance Dynamics Properties* (PDP). The temporal raw performance data is discarded after the analysis is completed. When the results of the next chunk are available, the two sets of found PDDs are merged. In the following writing we describe each element of the scheme in more details.

5.3 Dynamic Phase Profiling

In this section we give a formal definition of the Dynamic Phase Profile and the profile time-series. The brief introduction and the history of the method was given in Section 2.4.1.

Performance profiling is a technique for collecting and aggregating statistics of performance related data within a context defined by the applied instrumentation. The context could be also seen as a point in the space given by the instrumentation dimensions (e.g. MPI process, OpenMP thread, code region). The performance data is further classified into metrics. Each metric has a particular semantic, for example: execution time, number of messages and others. In our work we consider only non-decreasing metrics and their derivatives, which is a limitation of the profiling technique itself and not the techniques proposed in this thesis. First we give a formal definition of the common profiling technique and then extend it for the dynamic one.

Let C be a set of all instrumented source code locations, P a set of processes used to execute an application and M a set of performance metrics measured by a performance analysis tool. Then the *Static Profile* is a mapping of a particular source code location $c \in C$ executed on a process $p \in P$ onto a sum of metric values v_n of a metric $m \in M$ measured N times during the measurement period:

$$StaticProfile : (c \times p \times m) \mapsto \sum_{n=0}^{N} v_n, \tag{5.1}$$

The Static Profile is convenient to find a metric value (e.g. communication time) observed during the execution of a particular source code location on a particular process. However, this mapping does not include any information about the temporal behavior of the performance since the sum operation shrinks the time dimension.

Dynamic phase profiling utilizes the iterative nature of scientific applications, which is typically represented by a *progress loop*[29], e.g. the loop over simulation time steps.

Therefore, the dynamic phase profiling produces a separate profile for each iteration of the progress loop. Let $\{1, 2, ..., N\}$ be the set of progress loop iterations, then *Dynamic Profile* is defined as an extension of (5.1)

$$DynamicProfile : (c \times p \times m \times i) \mapsto \sum_{n=0}^{N_i} v_n, \tag{5.2}$$

where in addition to a source code location and a process, we map a progress loop iteration $i \in I$ onto a sum of metric values v_n of a metric $m \in M$ which was measured N_i times during the iteration i and is called *dynamic profile sample*. For simplicity, we will denote $\sum_{n=0}^{N_i} v_n =: v_i$, since we do not distinguish single readings v_n beyond the granularity of one iteration of the progress loop.

The technique proposed in this work focuses on the analysis of performance changes along the iteration dimension I independently of other three dimensions. Therefore, we simplify the mapping above by fixing the source code location c, process p and metric m and represent dynamic profile only as a function of i:

$$DynamicProfile_{c,p,m} : (i) \mapsto v_i, \tag{5.3}$$

Taking into account that the domain of the new mapping (i.e. iteration) is equidistantly sampled and represents time, we can denote dynamic profile as a *time-series*.

5.4 Online Access Interface

The OPDA scheme achieves constant time-independent size of temporal performance data by separating the processes of measurement collection and measurement analysis. This requires the monitoring library and the RAA to be able to exchange information.

We describe the part of the analysis scheme allowing this on the example of the Score-P performance measurement system which was extended with the Online Access Interface (OAI) for this purpose. The Score-P OAI, which is part of the measurement system, enables RAA to connect to the Score-P over TCP/IP sockets and to operate the measurement process remotely.

The part of the application execution for which performance measurements can be configured through the OAI interface is called online phase. The online phase has an associated user region containing the part of application source code which is of interest for the analysis and therefore has to be marked manually by the user with the provided preprocessing directives. In order to benefit from multi-step measurements, this region should be an iteratively executed part of the code (e.g. the body of the progress loop) with the potential for global synchronization at the beginning and at the end. Each phase region will become a root for a call-tree profile during one measurement iteration. Data exchange

with the RAA takes place at the beginning and at the end of the phase, thus it does not affect the measurements within the phase.

The communication with the RAA is done over TCP/IP sockets using a text-based monitoring request interface language which is a simplified subset of the request language used by Periscope. The syntax of the language covers a broad range of online analysis scenarios by means of three kinds of requests:

- Measurement configuration request,
- Execution request, and
- Measurement retrieval request.

The first category of requests allows enabling or disabling of performance metrics available in Score-P. The scope of enabled metrics is global, i.e. they are measured for every region within the online phase. Also some measurement tuning adjustments like depth limits for profile call-trees or filtering of high-overhead regions can be done with these requests. Execution requests are used to control multiple experiments by ordering Score-P to run to the beginning or to the end of the phase or, if the analysis is done, to terminate the application. Measured performance data, stored inside the Score-P call-tree profile, can be accessed by means of measurement retrieval requests. The profile data can be returned to the RAA in two ways: as a call-tree profile, where each node represents one call-path of the source code region with associated measurements attached, or as a flat profile, where measurements performed on some source code region are aggregated regardless of the call-path.

In addition to allowing scalable analysis of temporal performance data, the Score-P OAI has further benefits:

- Possibility for multiple experiments within one application run
- Avoiding dumping all measurements to a file at the end
- Faster measurement process: one iteration of the application could be sufficient
- Monitoring configuration refinement based on already received measurements

5.5 Online Processing of Temporal Performance Data

Figure 5.2 depicts online processing of temporal performance data in more details. The scheme can be broken down in three major parts:

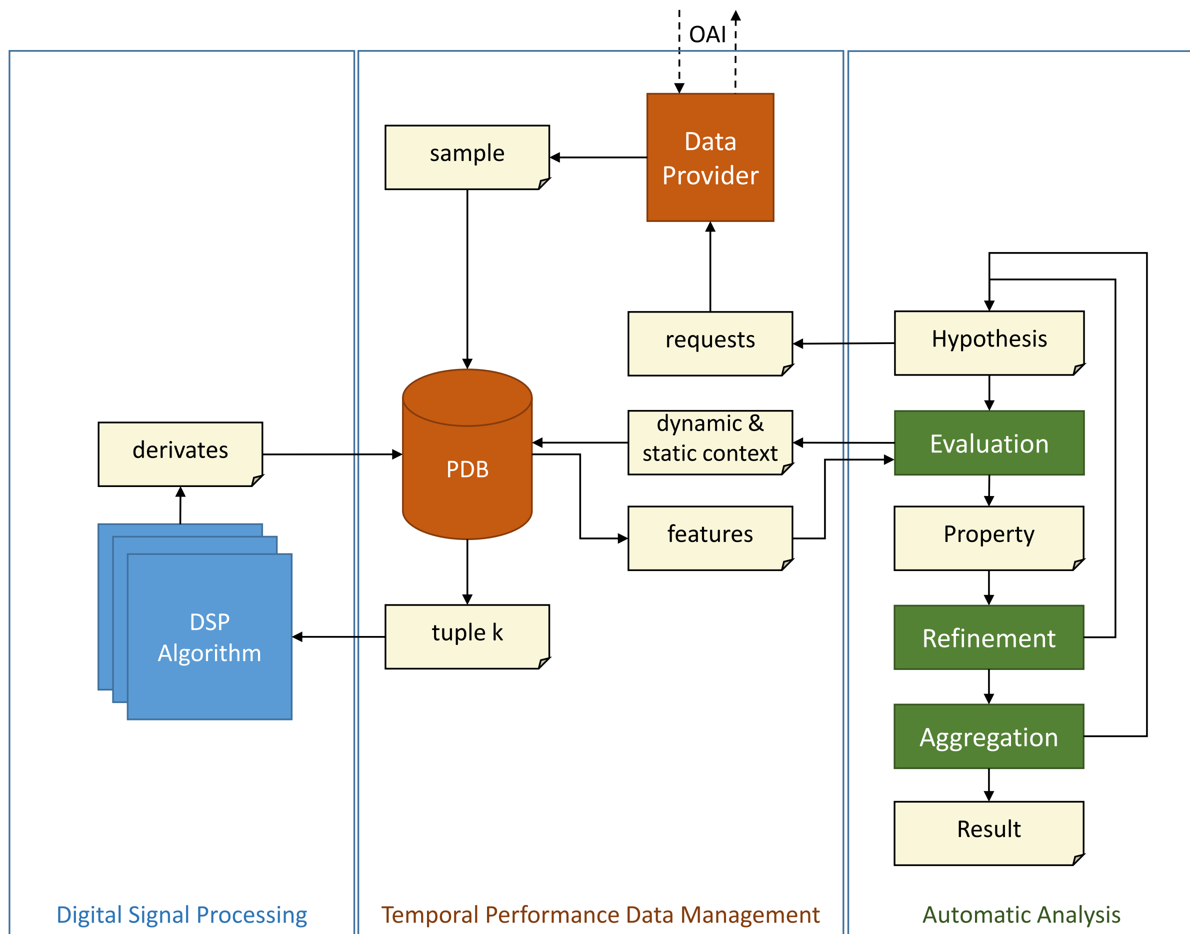


Figure 5.2: Online temporal performance data processing scheme.

- Temporal Performance Data Management
- Digital Signal Processing
- Automatic Analysis

The Analysis sub-system, shown with dark-green, defines the logic of the online automatic performance dynamics analysis. The algorithm is following a hypothesis-driven rule-based search for performance degradations.

Each hypothesis requests the raw performance data which it requires for the evaluation, e.g. MPI wait-state measurements. They also specify the length of the experiment in the iterations of the progress loop. Consequently this defines the size of the analysis chunk against which the hypothesis will be evaluated.

While the experiment is running the temporal performance data is being received, stored and manipulated by the data management sub-system drawn with the brown color. It

includes the Data Provider (DP) - a module which is responsible for accepting data arriving over the OAI. After a sample of the dynamic profile is received it is stored in the Performance Data Base (referred to as PDB). DP also provides an interface for accepting measurement requests from the Analysis sub-system which are then forwarded to the monitoring library over OAI.

However, the time-series of profiles received from the monitoring library can not be used to evaluate the hypothesis directly. Instead the evaluation rules are defined in terms of high-level features of the signal, e.g. the hypothesis might test the presence of a peak in the time-series. Therefore, the features have to be first extracted from the raw time-series data.

The features required by the hypothesis are defined on a Dynamic Context (DC). It specifies a subset of the time-series samples where the feature has to be computed and an interpolation algorithm to be used to handle missing values.

Depending on the static context, a chunk of profile samples is queried from the PDB and passed to the Digital Signal Processing (DSP) module. The module consists of a set of advanced algorithms for time-series analysis that implement requested transformations. After applying the transformation, the features are computed and returned to the analysis module for the evaluation. Although, the raw data is discarded after its analysis is completed, some derived statistics and transformations (in order to prevent temporal scalability bottlenecks these have to grow sub-linearly with respect to time) are preserved for the purpose of merging analysis results over multiple chunks or to prevent recalculating the same transformation twice.

Depending on the evaluation results the hypothesis can be accepted or rejected. In the first case we say that a performance property was found. In the next step the property can be refined into a set of new hypotheses. Those are then evaluated against the current chunk and, if needed, an additional burst of experiments can be requested.

The found properties however are only valid for the chunk where they were detected. Therefore, when the next chunk is analyzed and a new set of properties is found they have to be merged with the properties found in the previous step. After all chunks were analyzed the resulting set of properties constitutes the final report.

5.6 Improved Periscope Analysis Engine

In this section we present an implementation of the Online Dynamic Profile Analysis scheme in Periscope. The implementation adds new capabilities and ensures temporal scalability of the analysis.

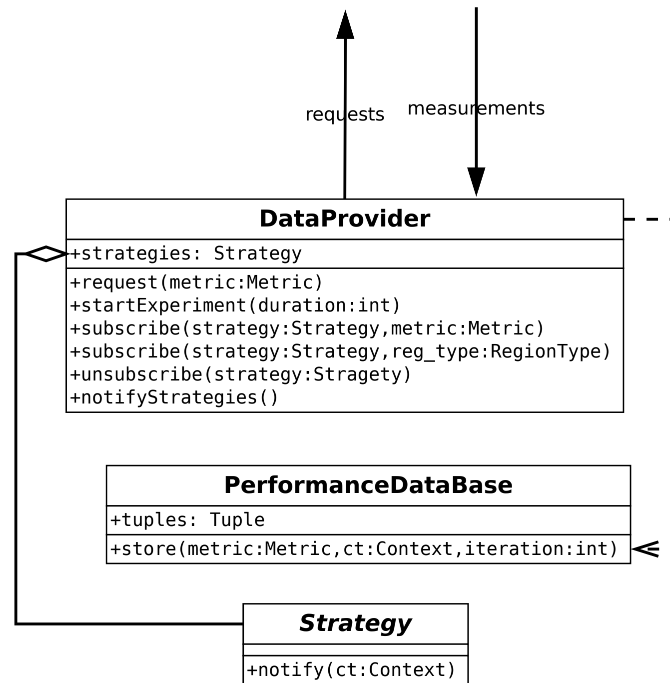


Figure 5.3: Data Provider class diagram.

5.6.1 Requesting and Storing Temporal Performance Data

As described above, the task of requesting and receiving temporal performance measurements on the RAA side is carried out by the Data Provider. The class diagram of it is presented in the Figure 5.3.

Data Provider exchanges messages with the monitoring library over OAI, where requests for measurements are sent and temporal performance data is received. After being received and processed each sample is stored in the Performance Data Base into the corresponding chunk of samples.

In order to inform the consumers of the measurements, in our case analysis strategies, DP implements the observer pattern. Following the pattern DP provides a subscription mechanism where an observer, i.e. strategy, can register for a specific measurement specified either by the metric type or the region type where measurements were collected. Although other ways to specify the measurements of interest are possible, e.g. by a measurement context, in practice we found the two filtering criteria above practical. While registering, a pointer to the observer is given to the DP and stored in the list of observers

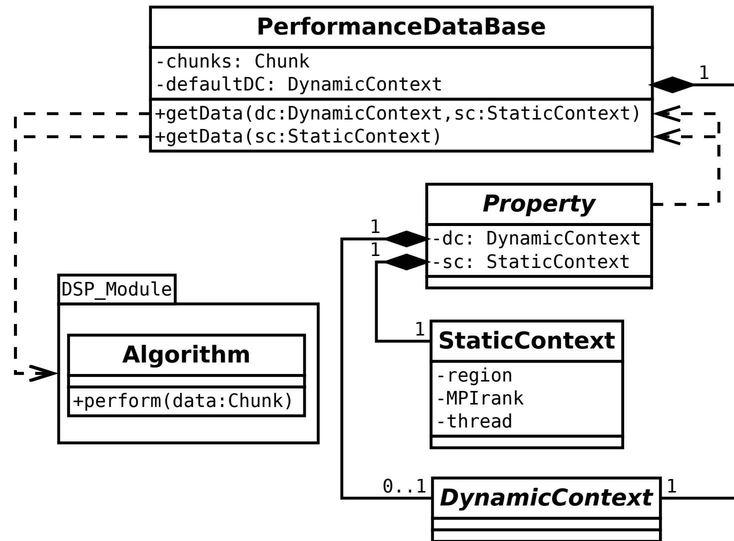


Figure 5.4: Performance Data Base class diagram.

to be notified when the requested metric is received from the monitoring library.

5.6.2 Accessing Temporal Performance Data

Temporal performance data is stored and managed by the Performance Data Base. The way the data is accessed is presented in the class diagram shown in Figure 5.4. Consumers of the data stored in PDB are properties represented by the *property* class. Each property evaluates a hypothesis within a specific context which is represented by the *staticContext* class. It is a composition of the source code location, i.e. region, and an execution location, i.e. MPI rank and OpenMP thread. Together with a metric type the static context uniquely identifies a dynamic profile time-series.

In case of properties evaluating performance dynamics, however, the evaluation rules are defined in terms of high-level features. These are computed from the raw temporal data using a set of signal processing algorithms discussed in the next chapter. The input raw data and initial preprocessing, such as interpolation, for the analysis are specified in the *dynamicContext* class. Properties use the dynamic context as a query to get the necessary

data from the PDB.

5.6.3 Backward Compatibility with Legacy Properties

Periscope contains a large collection of formalized performance inefficiencies in form of legacy properties. These test the presence of a performance problem in a specific static context. The temporal dimension is, however, completely ignored in this case.

In order to preserve this analysis logic we create one instance of the dynamic context class to be used as the default dynamic context by the PDB. It specifies the default operation to be applied when the dynamic context is not provided by the property. Since the properties evaluate hypotheses over the whole time domain, the default operation shrinks the time dimension by taking one of the following statistics over the samples:

- Sum
- Average
- Standard Deviation

Therefore, there are two ways to request data from the PDB. In the first case both static and dynamic context are provided. In the second case only the static context is provided by the property and the default dynamic context is used to perform the necessary aggregation to shrink the time dimension. This way we achieve backward compatibility with the legacy properties and preserve an extensive collection of formalized bottlenecks specifications.

5.6.4 Handling Missing Values

When dynamic profile is performed it can be the case that a particular region is profiled in one iteration, however, in the other one it might be not executed resulting in a missing value. Alternatively, due to an online measurement configuration a metric might be requested only in a subset of iterations which results in missing values as well. In order to apply meaningful analysis to this data, missing values have to be first properly interpolated. Following interpolation methods were implemented:

- Padding with zeros
- Piecewise constant interpolation
- Linear interpolation

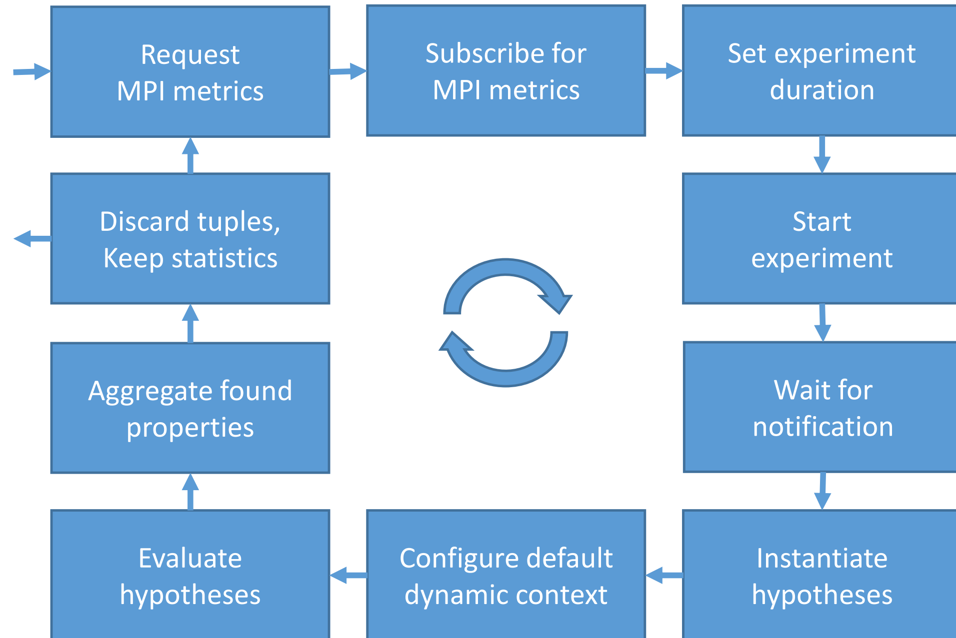


Figure 5.5: Simplified online analysis process flow diagram for the MPI wait-states analysis strategy.

The type of the interpolation to be used depends on the specific analysis configuration defined in the dynamic context. For example, when missing values are a result of a region not being executed in some iterations, then the zero padding interpolation is appropriate. On the other hand, when requested metrics cannot be measured simultaneously, e.g. due to limited number of counters, they can be multiplexed by measuring switching subsets of metrics every iteration. In this case the measurements can be interpolated using constant or linear methods. Of course, one has to be aware of possible aliasing effects due to down-sampling followed by the up-sampling interpolation.

5.6.5 Online Analysis

In this section we describe the online analysis process on the example of the MPI wait-states analysis strategy. Although, the strategy doesn't target dynamic characteristics of the performance, we use it as a simple example to demonstrate the main steps in the online processing of dynamic profile chunks measured over multiple iterations. Figure 5.5 presents a simplified analysis flow chart diagram of the strategy highlighting the main activities which are discussed below.

Request MPI metrics. In this step, the analysis strategy configures the collection of MPI metrics including wait-states by placing corresponding requests to the DP.

Subscribe for MPI metrics. After requesting the metrics, the strategy also subscribes for all MPI measurements to be received from the monitoring library at the end of the experiment. A call back handling each measurement separately is given to the DP.

Set experiment duration. Here the experiment duration in iterations of the application's progress loop is configured. In this example the duration is not relevant form the analysis point of view, but it is important in terms of limiting the size of temporal performance data collected and stored on the RAA side.

Start experiment. The experiment configurations set above are now transferred to the monitoring library over the OAI and the application is released. At the end of each progress loop iteration collected measurement samples are sent to RAA and stored in PDB. The experiment is run for the amount of iterations specified in the previous step.

Wait for notification. While the experiment is running the strategy is waiting for the notification about metrics it subscribed for. The notifications are issued after the experiment is completed.

Instantiate hypotheses. When the MPI wait-states are detected by the monitoring library and their duration is returned as a series of measurements, the strategy is notified about each static context where a particular type of wait-state was detected. For each reported static context a wait-state bottleneck hypothesis is instantiated.

Configure default dynamic context. The hypotheses instantiated above verify that the total wait time within a static context is above a certain percentage of the overall time. However, the data stored in the PDB is a series of measurements of the detected wait-states. Therefore, the series has to be reduced first. This is done by setting the default dynamic context to perform summation reduction operation over the chunk. Also, the interpolation method is selected to fill missing values with zeros since these can be only the case when corresponding regions were not executed during the execution.

Evaluate hypotheses. During this step the hypotheses are evaluated against the values specified by static context and aggregated according to the dynamic context configured above. If the accumulated wait time exceeds a certain fraction of the accumulated runtime of the iterations in the experiment, the hypothesis is proven and a found performance property is recorded.

Aggregate found properties. Here the properties found in the current experiment are merged along the time dimension with the properties found in previous experiments. In order to be merged, the properties have to be of the same type and have to be reported for the same static context. Different merging schemes are possible and depend on the property. In case of wait-state properties a simple summation of the wait-state durations in previous and current experiments is performed. It is then again compared against the sum of the execution times of the two experiments. Another alternative would be to store severities for each experiment and then perform a coarse grain analysis of the temporal evolution of the wait-states.

Discard raw data, keep statistics. In this step the raw temporal performance data used for the analysis is discarded. Nevertheless, some statistics like total execution time are preserved in particular for the properties aggregation process discussed above. After this step, the analysis cycle is repeated or terminated in case the application has finished or the requested analysis duration exceeded a given threshold.

5.7 Summary

In this chapter we presented a novel technique for online collection and analysis of dynamic profiles called Online Dynamic Profile Analysis. It separates the collection and the analysis of the temporal performance data between the monitoring library and the remote analysis agent interconnected by the Online Access Interface. The data is produced and processed on the fly which allows time-dimension-scalable low-overhead dynamic profiling and online analysis. This means that the amount of data to be buffered is decoupled from the number of measurement iterations on both sides. The proposed technique enables efficient performance dynamics analysis discussed in the next chapter.

Chapter 6

Automatic Analysis of Performance Dynamics

6.1 Motivation

In the previous chapter we proposed a scheme for the time-dimension-scalable collection and processing of temporal performance data. The problem statement for this chapter is how to extract relevant knowledge out of resulting numerous time-series of performance data? A fundamental problem here is to bridge a semantic gap between the raw temporal data and the desired high-level knowledge about performance dynamics that would be native to the "mental" model of the user.

The traditional approach to the analysis of such temporal data utilizes various graphical representations (1D-, 2D-, 3D-plots, histograms and so on) and then lets the user to "bridge the gap" on her own. Having an infinite variety of all possible realizations of the dynamic process reflected by the temporal data, it is still remarkable how well the humans are capable of interpreting this kind of representations. Nevertheless, we go one step further and automate this process.

One would ask then, why should we care of automating the process of interpretation of temporal data when humans already have outstanding capabilities for such kind of tasks? The reason is that with both the length and the number of time-series growing with the measurement time and the size of the experiment respectively, the effort of manual analysis grows proportionally. This makes manual interpretation overwhelming when studying the performance dynamics of real-world long-running applications on current or future HPC systems.

In this work we aim at automatically detecting performance optimization potentials in the application execution process represented by time-series of performance measurements. This process is a combination of multiple factors, i.e. cache effects, load-balancing, external interrupts, etc., which manifest themselves at different time-intervals and time-scales.

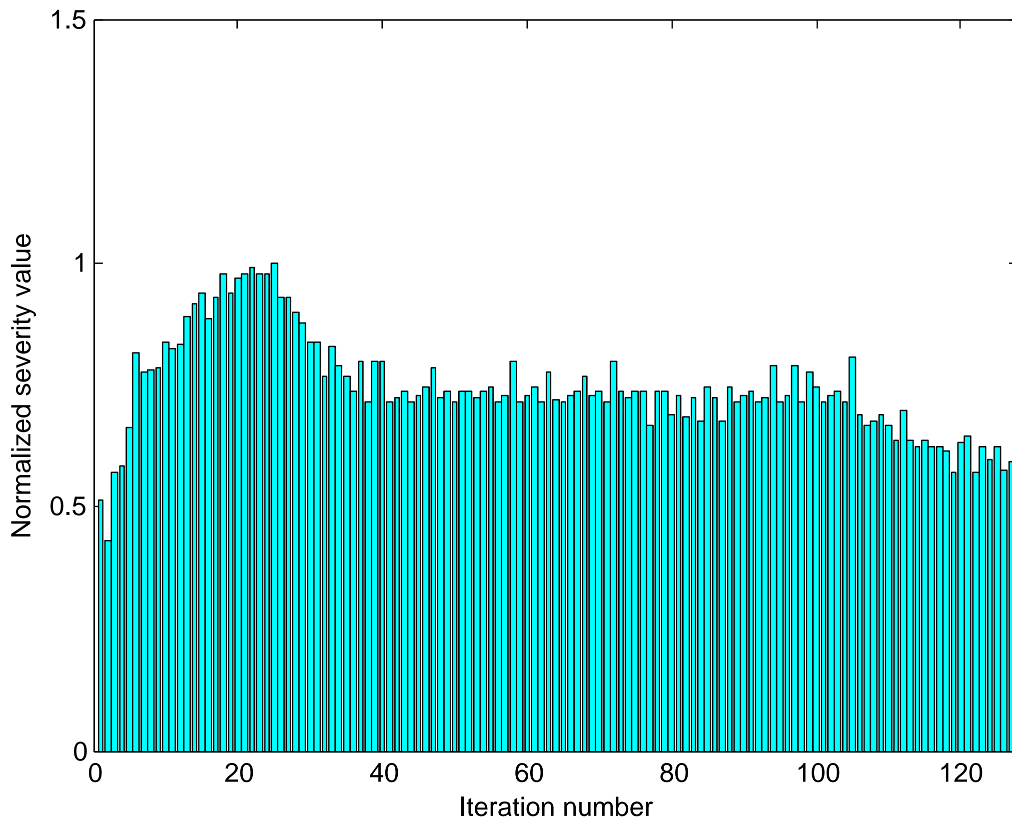


Figure 6.1: Example signal plotting severity values of the property "Hot Spot of the Application" for 128 iterations of the CX3D progress loop. The values are normalized to the $[0, 1]$ interval.

These result is either worsening or improving transient performance trends. Therefore, the goal of the performance dynamics analysis is to identify, i.e. localize in space and time, and to communicate to the user negative trends which in the following writing will be referred to as *degradations* for brevity reasons.

6.1.1 Example Signal

The techniques we are using to achieve our goals are from the field of signal processing, therefore, we us the term *signal* to denote a time-series of performance measurements.

In order to simplify understanding of the techniques, we demonstrate them on an example signal collected in a performance dynamics experiment with a simulation code called CX3D. The application is used for simulations of the Czochralski crystal growth [44] in the silicon wafer production. The signal is shown in the Figure 6.1.

On the y-axis absolute severity values of the "Hot spot of the Application" performance property are given. This type of property evaluates a percentage of the CPU time spent in a region for which the property is evaluated. Therefore the absolute severity value is nothing else as the CPU time of the region. The values shown in the figure are evaluated for the `velo` subroutine implementing the solver of the application and are normalized to the $[0, 1]$ interval.

Nevertheless, the semantics of the signal is not essential. The algorithms we propose in this chapter are designed to be agnostic to the performance metric being analyzed. What is important is that, since the y axis denotes severity of a problem, we denote any increases in the value as a degradation. The x-axis is represented by the iterations of the progress loop in which the property was evaluated.

6.1.2 Design Goals

As mentioned above, our goal is to automatically detect and communicate performance degradations to the user. We achieve this by automating the analysis process employed by the user, namely we aim at automatically answering the typical questions asked by the user when evaluating time-series of performance measurements:

- Does the performance change with time?
- What are the relevant performance degradations?
- When and where do they happen?
- How severe are they?

Quantify variability. The technique should be able to provide a measure of variability observed in a signal. Using this measure we should be able to compare the amount of variability between two signals. Also, since the temporal data we are working with is not stationary, we should be able to quantify variability locally within sub-intervals of a signal. In respect to the example shown in Figure 6.1, the answer to the first question would be the result of the comparison of the variability measure against a given threshold. Also one can observe that the variability in the first half of the signal is higher than in the second one.

Declarative qualitative representation. In order to answer the second question we need a representation model similar to the "mental" model employed by the user. For example, consider a representation in terms of a polynomial fitted to the example signal:

$$f(x) = p1 * x^5 + p2 * x^4 + p3 * x^3 + p4 * x^2 + p5 * x + p6, \tag{6.1}$$

where $p1 = 1.042e-09(8.71e-10, 1.213e-09)$, $p2 = -3.795e-07(-4.35e-07, -3.241e-07)$, $p3 = 5.03e-05(4.38e-05, 5.679e-05)$, $p4 = -0.002921(-0.003253, -0.002589)$, $p5 = 0.06763(0.06065, 0.07461)$, $p6 = 0.4032(0.358, 0.4483)$. Although, capturing the dynamic behavior of the signal, such representation gives very little insight to the present degradations when reported to the user explicitly.

Instead, we need a representation which is formulated in high-level primitives (printed in italic below) which are declarative and readable by humans, such as:

"Signal has a *peak* near iteration 20, then it is followed by a *flat interval* until iteration 100, where it shows a *step decrease*."

Quantify relevance.

The answer given above is clearly not complete, i.e. it doesn't describe all transient features present in the signal. When looking more precisely one can recognize numerous small spikes or a local minimum around iteration 85. However, if we report every feature to the user, it will be no better than presenting the raw data and again letting the user decide what is relevant. Therefore, the technique should be able to quantify relevance and report only those features, which would be visually found as relevant by the user in a manual analysis.

Multi-scale analysis.

The execution process, which performance dynamics we analyze, is a combination of numerous factors which manifest themselves at different time intervals and scales. For example, a sudden mesh refinement can happen in a certain iteration resulting in a step increase of synchronization losses due to a worse load balancing. On the other hand, wait-states propagating over point-to-point communications and accumulating over several iterations will result in gradual increase in corresponding measurements. This argues for a systematic multi-scale approach for searching relevant features among all scales, since there is no preferred one. Additionally, the technique should be able to locate features in time in order to answer the third question.

Quantify severity. After detecting and localizing relevant features we need to quantify the severity of a particular feature in respect to the overall performance. This value puts a maximum limit on the performance improvement which can be gained in case the degradation is prevented by optimization.

6.2 Wavelet Analysis

Wavelet transform is a promising signal processing technique[43]. When compared to the more known Fourier transform, which analyses signals in the frequency domain, the Wavelet transform is better suited for the representation of sudden spikes and discontinuities capturing both temporal and frequency characteristics of the signal. Since such

patterns are common for the profile measurement time-series, this transform offers promising insights into performance dynamics.

6.2.1 Discrete Wavelet Transform

The discrete version of the Wavelet analysis is called Discrete Wavelet Transform (DWT). It decomposes a one-dimensional signal into a two-dimensional representation of shifted and scaled prototype versions of a bandpass wavelet function ψ with local support, called mother wavelet, and shifted versions of a lowpass scaling function ϕ also with local support. For this work, we select a basis built from Haar wavelet [43], which is better suited for representation of jumps in the studied signal.

Consider a discrete time-series $f[i]$, where $1 \leq i \leq N$, $N = 2^J$, J a positive integer, and sampling interval equals to 1. The wavelet basis is constructed by scaling a discrete mother wavelet ψ with dyadic scales $s = 2^j$, where $1 \leq j \leq \tilde{J}$ is the scale index, $\tilde{J} \leq J$ is the level of decomposition of lower scales; and shifting it with dyadic shifts $u = 2^j n$, where $0 \leq n < 2^{\tilde{J}-j}$ is the shift index. Higher scales $0 < k < 2^{J-\tilde{J}}$ are captured by the lowpass scaling function ϕ which is shifted with shifts $u = 2^{\tilde{J}}k$. Described above shifted and scaled wavelet functions $\psi_{j,n} = \frac{1}{2^{j/2}}\psi(\frac{n}{2^j})$ together with the shifted versions of the scaling function $\phi_{\tilde{J},k} = \frac{1}{2^{\tilde{J}/2}}\phi(\frac{n}{2^{\tilde{J}}})$ form an orthonormal basis. Then a time-series $f[i]$ could be represented in this basis as follows [43]:

$$f[i] = \sum_{j=1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} W_f[j,n]\psi_{j,n} + \sum_{k=0}^{2^{J-\tilde{J}}} L_f[k]\phi_{\tilde{J},k} \tag{6.2}$$

, where $W_f[j,n] = f \circledast \psi_{j,n}$ and $L_f[k] = f \circledast \phi_{\tilde{J},k}$, $\circledast$ denoting the convolution operator, are wavelet and approximation coefficients respectively.

The sum of squares of the time-series samples is usually referred as energy. An important property of the DWT is that the total energy of the original signal is equal to the sum of energies of the wavelet and the approximation coefficients [43]

$$\sum_{i=1}^{N} |f[i]|^2 = \sum_{j=1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2 + \sum_{k=0}^{2^{J-\tilde{J}}} |L_f[k]|^2. \tag{6.3}$$

Furthermore, for the Daubechies class of wavelet functions, in particular the Haar wavelet, the wavelet filter acts as a high-pass filter. Thus, by neglecting the $L_f[k]$ terms, it decomposes the sample variance of a time-series as [51]

$$Var(f) = \frac{1}{N} \sum_{j=1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2. \tag{6.4}$$

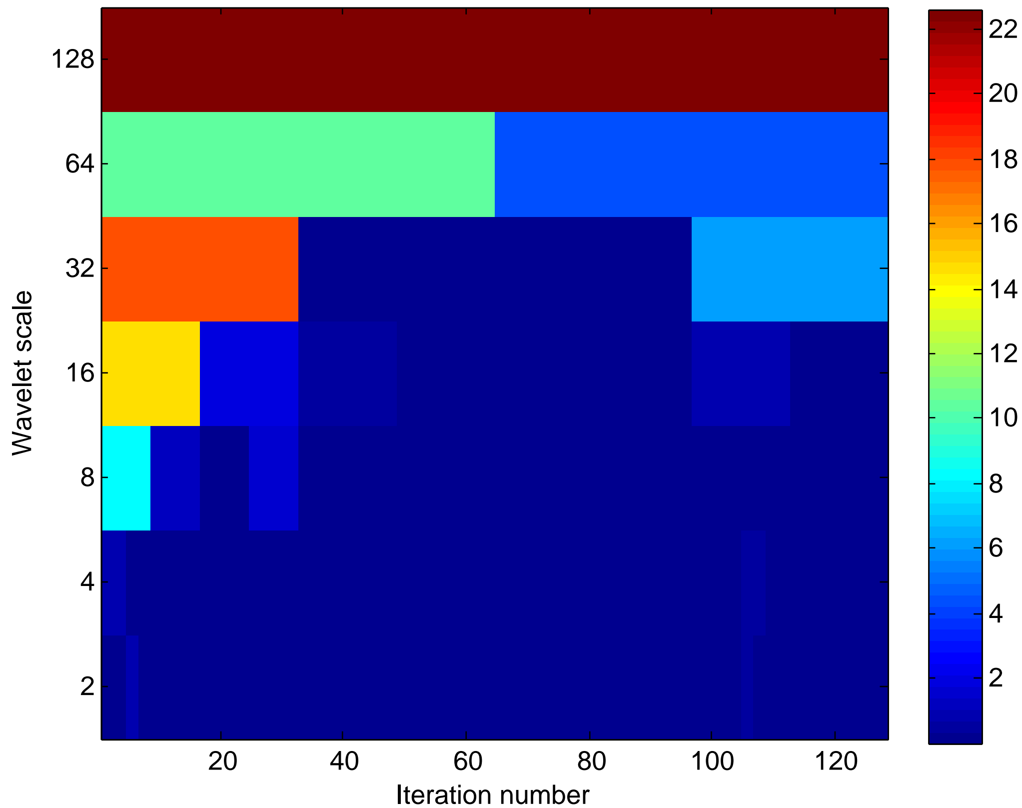


Figure 6.2: Scaleogram of the example signal. It shows the energies of the wavelet coefficients in percentages to their sum plotted against corresponding scale and temporal location.

Figure 6.2 shows the Scaleogram of the example signal. Scaleogram is a plot of the energies of the wavelet coefficients (here we plot percentages of the sum) against the scale, y-axis, and the shift, x-axis. Mention the light-red rectangle at the scale 32 in the left part of the plot. It corresponds to the high energy values which are due to the peak around iteration 20 in the example signal. Also, as expected, all the high-intensity coefficients are in the first half of the signal meaning that we have more variability there.

6.2.2 Implementation in Periscope

In our analysis DWT is used to quantify the variability observed in the signal. Furthermore, it allows to calculate the amount of variability at different scales and temporal locations denoted by the shift index.

We extend Periscope by implementing the DFT as part of the signal processing module. After decomposing the signal into wavelet coefficients, the following functions can be used

to get variability measures:

- **getTotalEnergy()** - returns the total energy of the signal which is equal to the sum of squares of the signal samples, or, due to the energy conservation property of the DWT, to the sum of the wavelet and the approximation coefficients, $totalEnergy = \sum_{j=1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2 + \sum_{k=0}^{2^{J-\tilde{J}}} |L_f[k]|^2$.
- **getDynamicEnergy()** - returns the sum of energies of the wavelet coefficients, called dynamic energy, $dynamicEnergy = \sum_{j=1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2$.
- **getShortScalesEnergy()** - returns the sum of energies of the wavelet coefficients belonging to the first half of scales, $shortScalesEnergy = \sum_{j=1}^{floor(\tilde{J})/2} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2$. We use this to quantify variability due to shorter scale variability such as spikes.
- **getWideScalesEnergy()** - returns the sum of energies of the wavelet coefficients belonging to the second half of scales, $wideScalesEnergy = \sum_{floor(\tilde{J})/2+1}^{\tilde{J}} \sum_{n=0}^{2^{\tilde{J}-j}-1} |W_f[j,n]|^2$. We use this to quantify variability due long term variability such as trends.

6.3 Qualitative Representation of Trends

Given a time-series of dynamic profile samples, a direct meaningful interpretation of the raw values is hardly possible. Therefore, in order to automate the analysis of dynamic performance, we need first a representation model, allowing explicit and declarative representation of both quantitative and qualitative features which are native to the ”mental” model employed by the user. In this section we present such a representation model and a corresponding mapping which we adapt from the works of Backshi, Cheung and Stephanopoulos [16],[22] from the field of Chemical Engineering.

In order to represent and reason with temporal information, we need to represent time explicitly and concisely. In our case the iterations of the progress loop define the temporal dimension as a sequence of strictly increasing iteration numbers:

$$I = 0, 1, ..., i_n, ..., i_N$$

Consider a time-series of dynamic profile samples $f[i]$, then the *qualitative state* of f at $i \in I$ is defined as a triplet of boolean values as follows[22]:

$$QS(f,i) = \langle [f[i], f'[i], f''[i]] \rangle, \tag{6.5}$$

where

$$[f[i]] = \begin{cases} + & \text{if } f[i] > 0, \\ 0 & \text{if } f[i] = 0, \\ - & \text{if } f[i] < 0; \end{cases}$$

$$[f'[i]] = \begin{cases} + & \text{if } f'[i] > 0, \\ 0 & \text{if } f'[i] = 0, \\ - & \text{if } f'[i] < 0; \end{cases}$$

$$[f''[i]] = \begin{cases} + & \text{if } f''[i] > 0, \\ 0 & \text{if } f''[i] = 0, \\ - & \text{if } f''[i] < 0. \end{cases}$$

Using the definition above we define the *qualitative trend* of the time-series, $f[i]$, as a sequence of qualitative states over I.

Unfortunately, the values of the discretized function $f(i)$ are not available outside of I which makes exact calculation of its derivatives in I not possible. Therefore, we must accept approximations of those, which are obtained by means of numerical differentiation.

Nevertheless, such approximations allow representations with increasing abstractions which are very close to intuitive notions employed by humans in interpreting the temporal behavior of functions. Indeed, the way we were taught to sketch a function in the calculus is by segments between two nearest zero-crossings of the function or its first two derivatives. In our setup it corresponds to segments of constant qualitative states. Therefore we extend the notion of qualitative states to qualitative episodes in order to emulate this intuitive representation:

Let the qualitative state $QS(f,i)$ be constant $\forall i \in \tilde{I} \subseteq I$. Then the *qualitative episode*, E, of f over $\tilde{I}$ is the pair $\langle \tilde{I}, QS(f,\tilde{I}) \rangle$, with $\tilde{I}$ explicitly specifying the temporal extent of the episode and $QS(f,\tilde{I}) = QS(f,i) \forall i \in \tilde{I}$ characterizing the constant qualitative state over $\tilde{I}$.

Whenever two episodes, defined over adjacent time intervals, have the same qualitative state values, they can be merged to form an episode spanning over the union of both temporal extents. On the other hand, if the qualitative state is constant over an interval, then it is also constant over any of the subintervals. Using this observation we define a *maximal episode* as an episode E, such that there is no episode E' for which the qualitative state is equal to the one of E and the temporal extent of E is contained in the temporal extent of E'. From these definition we conclude, that two maximal qualitative episodes are separated by a change in qualitative state, i.e. a critical point of the function.

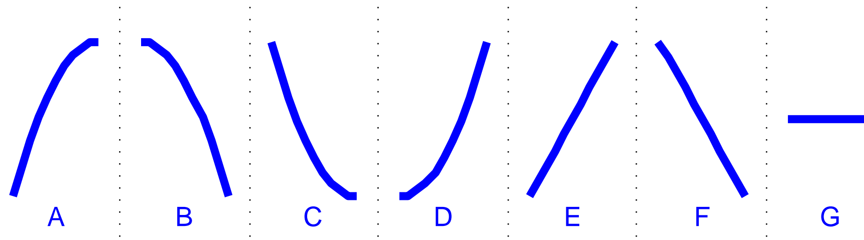


Figure 6.3: Geometric primitives for the 7 basic qualitative descriptors of the qualitative representation language. A: concave increase, B: concave decrease, C: convex decrease, D: convex increase, E: linear increase, F: linear decrease, G: constant

6.3.1 Geometrical Interpretation

Although, the qualitative representation formalized above matches the intuitive primitives used by humans, it is not explicitly declarative when presented in the form (6.5). Instead, we use a language that offers declarative geometric semantics.

As one can see the maximum number of combinations of (6.5) is equal to 27. After removing all non-valid combinations such as when $f(i) = 0$ and $f'(i) \neq 0$ and $f''(i) \neq 0$; when $f'(i) = 0$ and $f''(0) \neq 0$ and the combinations when $f(i) < 0$, which are not possible since our metrics are all positive, we are left with only 7 possible combinations. We label each valid combination with an alphabetic character and a geometric primitive visualizing the combination of signs of the first and second derivative by the corresponding curve. These are shown in Figure 6.3.

As an example, consider a qualitative representation of the smoothed (in the next section we explain why we choose to smooth the signal before extracting the qualitative states) example signal shown in Figure 6.4. Here we plot qualitative episodes detected in the signal depicted by both alphabetic labels and corresponding basic shapes adjusted to the width of the detected qualitative episodes. One can mention that the resulting representation is very close to the intuitive perception of the dynamic behavior of the signal.

6.3.2 Scale-Space Filtering

As shown in the previous section, qualitative representation by means of the language of qualitative episodes provides a compact and intuitive description for dynamic features. The episodes are formed by extrema in the signal and its first two derivatives. These, in their turn, require computation of the derivatives which have to be taken over some neighborhood. The extent of the neighborhood is often referred as to *scale* and choosing it properly appears to be a fundamental problem since there is no rule for preferring one

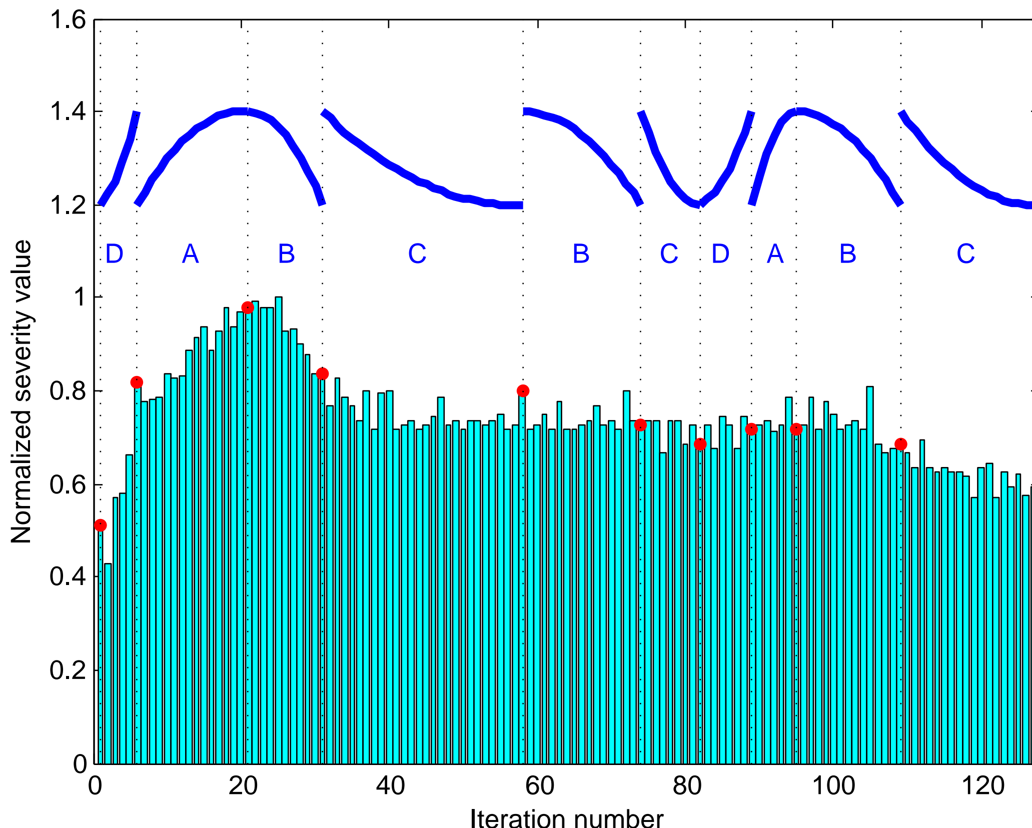


Figure 6.4: Qualitative representation of the smoothed example signal. The original signal is plotted with cyan bars; the zeros of the second and first derivatives of the smoothed signal are shown with red dots; the qualitative representation by both alphabetical labels and geometric primitives is plotted above with blue color.

against another. The reason is that the events that we find meaningful and important tremendously vary in size and extent and depend on the overall context. In this section we describe a systematic way for handling the scale parameter called *Scale-Space Filtering* (SSF) which was introduced by Andrew P. Witkin in [61].

6.3.3 Scale-Space Image

The approach to managing the scale parameter in SSF follows the idea that (1) the amount of details in the signal is decreasing and (2) no new details are allowed to appear as the scale is increasing.

As it was shown in [15] a smoothing by convolution with the Gaussian filter and varying the Gaussian standard deviation allows to achieve the desired behavior. The details being incrementally swiped away in this context are zero-crossings of the second derivative of

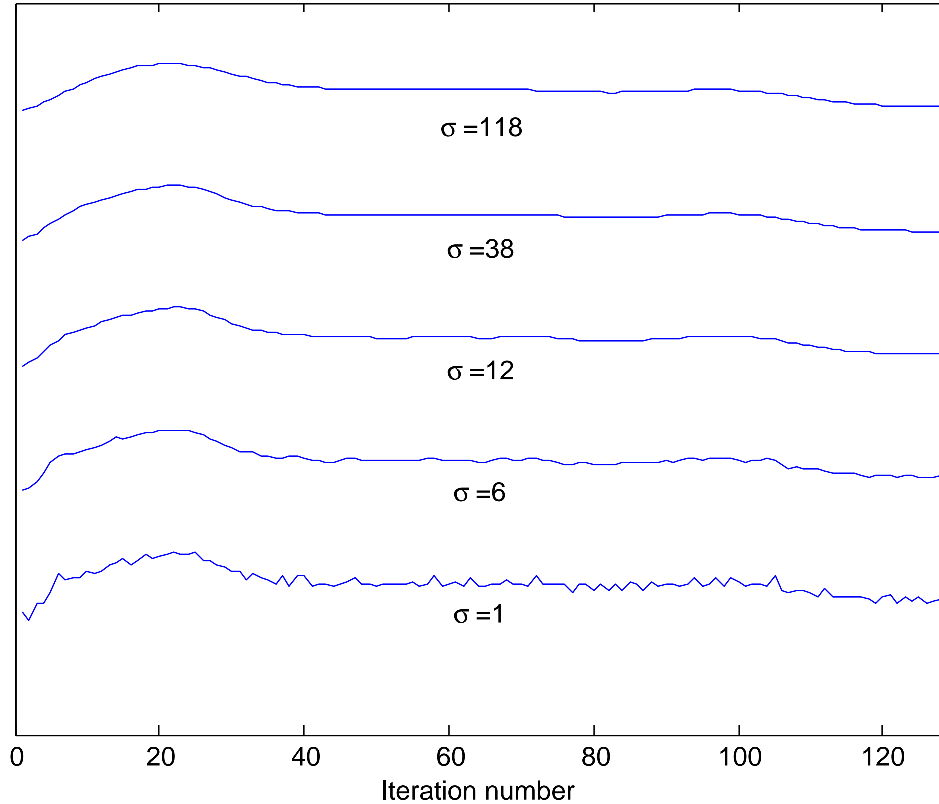


Figure 6.5: Slices of the example signal's Scale-Space Image at the given set of scales, σ.

a signal, i.e. inflexion points. The Gaussian convolution of a signal f depending both on argument x and the signal independent scale parameter σ, i.e. Gaussian's standard deviation, is given by

$$F(x,\sigma) = f(x) \circledast g(x,\sigma) = \int_{-\infty}^{\infty} f(u) \frac{1}{\sigma(2\pi)^{\frac{1}{2}}} e^{-\frac{(x-u)^2}{2\sigma^2}} du, \qquad (6.6)$$

where "$\circledast$" denotes the convolution operator with respect to x. Function, $F(x,\sigma)$, defines a surface on the (x,σ)-plane, where each value of σ is mapped onto a smoothed version of f and the degree of smoothing is strictly increasing with σ. Following the terminology in [61] we call the (x,σ) domain *scale-space* and the function, F, the *Scale-Space Image* (SSI) of f. Figure 6.5 shows a part of the scale-space image for a set of scales built from the example signal.

An important property of Gaussian smoothing is that as we decrease scale, additional zero-crossings can appear in pairs, as it will be demonstrated in the following writing, but no existing ones can disappear. Moreover, the Gaussian filter is shown to be the only one featuring this property [15].

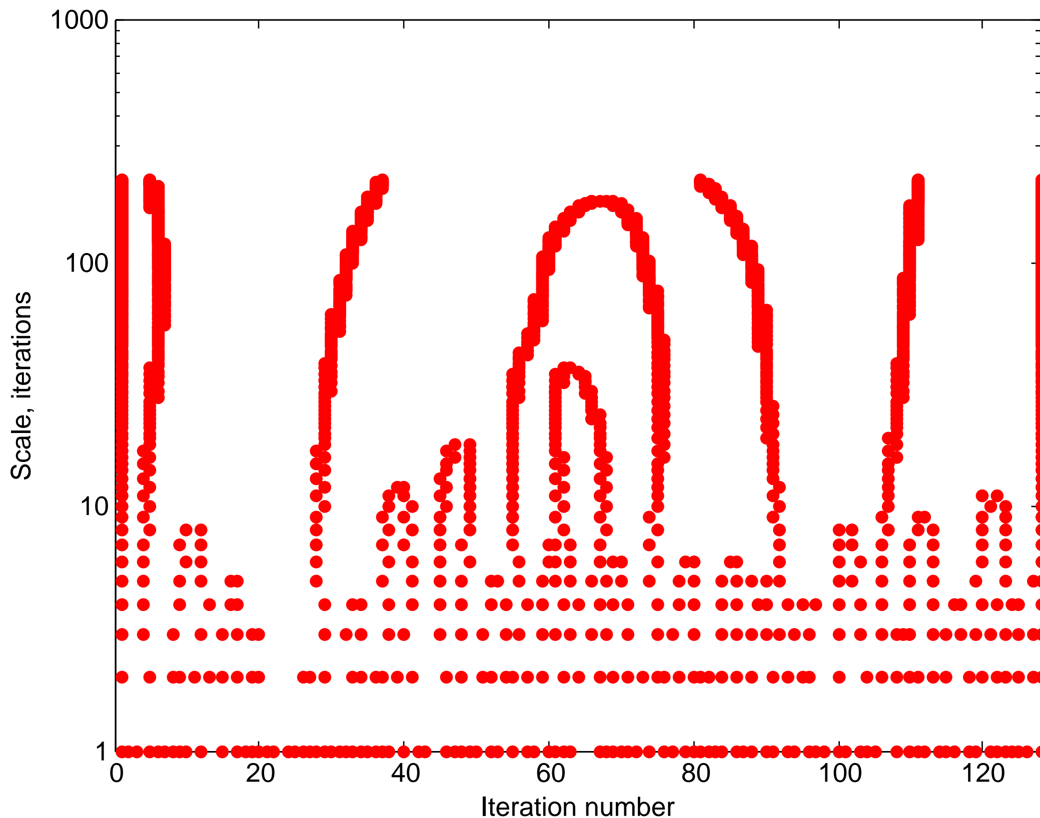


Figure 6.6: Contours of zero-crossings of the example signal's SSI.

6.3.4 Scale-Space Qualitative Representation

SSI is a multi-scale representation of a signal as a series of the smoothed versions of it. What we need, however, is a multi-scale representation in terms of qualitative episodes presented in Section 6.3.

An intermediate representation allowing such a transition can be obtained by connecting inflection points over scales of SSI resulting in *Contours of Zero-Crossings* (CZC). CZC for the example signal is shown in Figure 6.6

Here we can see a graphical evidence to the property of Gaussian smoothing that no new zero-crossings can appear with increasing scale. Additionally, from the close spatial proximity of zero-crossings belonging to the same contour we make an assumption that they arise from the single underlying effect. This allows tracking of signal's features over multiple scales. The contours of zero-crossings form arches which are properly nested in the spatial domain.

Another observation is that the contours are not vertical, i.e. the spatial location of an inflexion point is drifting with increasing scale which is a natural effect of smoothing.

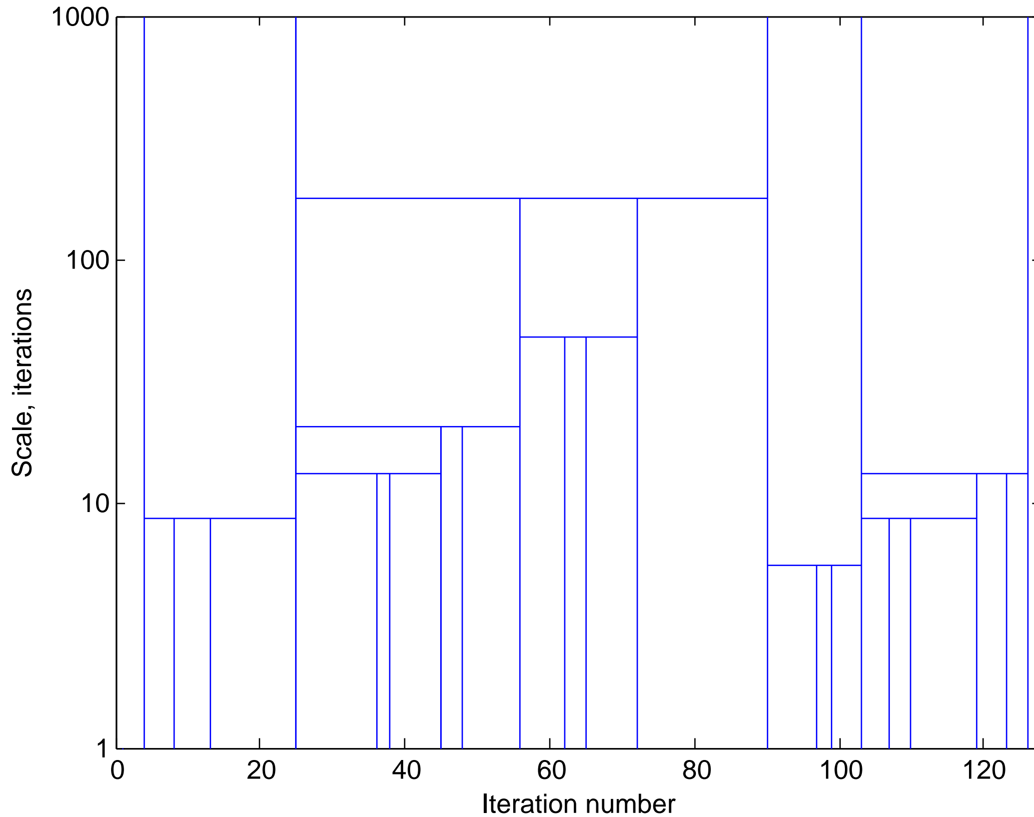


Figure 6.7: Interval tree of the example signal's SSI.

Using coarse-to-fine tracking of zero-crossings we can still obtain the precise location by taking the one from the finest scale. Therefore, we can represent each arch of a contour by the coarsest scale till which it survives smoothing and the precise spatial location obtained from the finest scale. Using this we redraw CZC by representing each arch of a contour by a vertical line starting at the zero-crossing's location at the finest scale and going to the coarsest scale where it gets connected by a horizontal line with the second arch. The transformed CZC, called Interval Tree, for the example signal is shown in Figure 6.7.

The hierarchical structure of zero-crossings, which we observed on the original CZC as a perfect nesting of contours, becomes prominent in the interval tree. This follows from the property of Gaussian smoothing stating that as we increase scale, additional zero-crossings appear in pairs at certain scale and then never disappear.

Each rectangle in the interval tree represents an interval between two neighbor inflexion points over a span of scales. With additionally detecting zero-crossings of the first derivative within this interval at the corresponding scales we label resulting sub-intervals with qualitative descriptors described in the previous section. Taking into account the hierarchical tree-like structure of the episodes, the result is the desired multiscale representation

concisely and completely describing the qualitative structure of the signal over all scales.

6.3.5 Qualitative Summarization

Looking at the interval tree one can mention that the vertical boundaries corresponding to the scale parameter σ vary for different episodes. The height of the rectangle corresponds to the number of scales over which the boundary inflexion points are present in the incrementally smoothed signal. Apparently, empirical observations indicate [61],[23],[17] that this value, called *stability*, corresponds to the perceptual salience of the corresponding qualitative episodes. In other words, the episodes with high stability tend to stand out among others with lower stability. As a remark supporting this statement, an interesting relation, although in a two-dimensional case, was shown between the spatio-temporal scale-space generated by Gaussian and the spatio-temporal receptive field response profiles registered from mammalian vision[41].

We use this property to quantify visual "relevance" of the signal's qualitative episodes, allowing to rank corresponding temporal features according to their perceptual significance. Furthermore, we shrink the multiscale representation to a single-scale one by performing summarization based on the stability value. The summarization algorithm descends the interval tree in a breadth-first fashion and selects the level with maximum value of the sum of stability values of the nodes at the level. This level is called *maximum stability level* (MSL). The sequence of the qualitative episodes given in the beginning of this section in Figure 6.4 corresponds to the nodes of the MSL of the example signal.

6.3.6 Implementation in Periscope

Numerical Implementation

The SSF definition given in 6.6 is for continuous signals. However, in order to be used in practical applications we need a discretized version of the SSF technique. For our implementation we use the scale-space theory for discrete signals developed by Tony Lindeberg [40]. Without diving into the details, which can be found in the original paper, we provide a short summary of the technique.

The straight-forward approach of sampling the Gaussian and discretizing the convolution integral, may lead to violations of scale-space assumptions, such as appearance of phantom zero-crossings. The reason is that the theory developed by Witkin [61] and for the 2D case by Koenderink [39] was developed for continuous signals and it is not guaranteed that the basic scale-space conditions are preserved.

Alternatively, a genuinely discrete scale-space theory is obtained by discretizing the diffusion equation under the postulation of the scale-space axioms. This allows to compute SSI of discrete signals by convolution with a family of kernels $T(n, t) = e^{-t} I_n(t)$, where

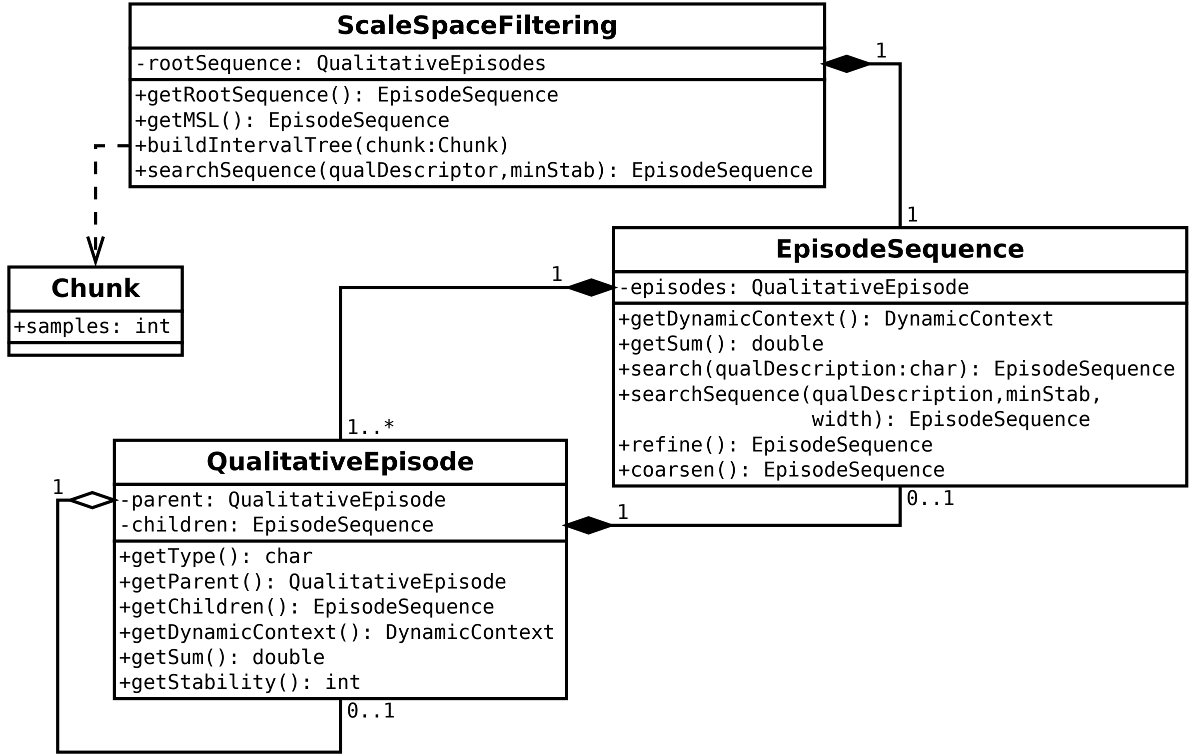


Figure 6.8: Class diagram of the SSF algorithm and related data-structures.

$t \in \mathbb{R}$ is the scale parameter and I_n are Bessel functions of integer order, n. These are shown to be the discrete analog of the Gaussian kernel, leading uniquely to the Gaussian kernel when similar argumentation applied in the continuous case.

Class Diagram

Figure 6.8 shows the class diagram of the SSF algorithm and corresponding data-structures. The `ScaleSpaceFiltering` class performs the actual processing of the raw temporal data stored in an object of the `Chunk` class. This includes smoothing, detection and coarse-to-fine tracking of zero-crossings as well as the classification of the qualitative episodes.

Each episode is represented by an instance of the `QualitativeEpisode` class, which stores the following meta-data:

- Type - the qualitative descriptor of the episode which is stored as a char
- Dynamic Context - the data-structure defining the borders of the iteration interval of the episode. These are the x coordinates of the enclosing critical points at the finest scale.

- Sum - sum of time-series samples within the episode collected at the finest scale
- Stability - the number of scales at which the episode exists
- Parent - the parent episode in the hierarchy of episodes defined by the interval tree
- Children - a sequence of episodes which are below the current one in the hierarchy

Sequences of episodes, such as children episodes of an episode or the maximum stability level, are represented by the `EpisodeSequence` class. It allows aggregation of episodes meta-data, as well as derivation of other sequences. Thus, one can refine or coarsen the current sequence by constructing a new sequence consisting of the children or parent nodes respectively. Furthermore, it allows to search for a subsequence of episodes specified by a sequence of qualitative descriptors as well as by constraints such as span, stability and so on.

6.4 Performance Dynamics Analysis Strategy

In the previous sections we presented algorithms that allow extraction of dynamic features from the raw temporal performance data. In this section we present a rule-based inference algorithm, called *Performance Dynamics Analysis Strategy* (PDAS), capable of automatic evaluation of temporal features and deriving high-level performance dynamics properties.

6.4.1 Design

The algorithms presented in the previous sections are capable of detecting dynamic features in time-series of performance data regardless of their semantic. The naive approach of applying them to all collected time-series of performance measurements will not lead to the desired results. For example, by processing a time-series of floating point operations completed within an iteration of the progress loop we might identify a spike around some iteration. How does this information help in identifying a potential for performance optimization? Or, what is the value of a knowledge about present degradations in temporal evolution of cache performance, when the average cache miss rate for the whole application execution is negligible?

Therefore, before applying the dynamics analysis, we first evaluate *runtime-static* performance of the application by means of available legacy analysis strategies of Periscope, here called as *Static Performance Analysis Strategies* (SPAS). As runtime-static performance we denote performance characteristics averaged over the application execution time, i.e. application performance profiles. Since the analysis algorithms are agnostic to the type of temporal performance data, any types of SPAS can be used, for example, MPI wait-state

analysis or OpenMP synchronization overheads analysis. Moreover an arbitrary number of SPASs can be used when the monitoring capabilities allow this.

The result of the analysis with a SPAS are runtime-static performance properties (SPs) which represent a particular performance problem, e.g. "excessive MPI wait time due to a late sender in MPI_Recv"; locate it in the space of application regions and execution units; and, finally, quantify the severity of it. Therefore, runtime-static properties answer questions "what" the problem is, "where" it is located and "how severe" it is.

By applying dynamics analysis to the time-series representing temporal development of the performance problem we answer the missing "when" question. This allows to locate the problem along all search dimensions. Moreover, our approach significantly reduces the search space for the performance dynamics analysis strategy as well as interpretation effort for the user by filtering out performance dynamics patterns when they are not relevant in the overall context of the application performance.

Therefore, there are three main phases in the performance dynamics analysis process:

- Analysis of runtime-static performance
- Expansion of the time dimension for the found runtime-static properties
- Analysis of performance dynamics

6.4.2 Analysis Algorithm

Analysis of Runtime-Static Performance

The algorithm of the PDAS is presented in Figure 6.9. The analysis starts by configuring a burst of measurement experiments of length n, following the concept of Online Dynamic Profile Analysis described in Section 5.2. Afterwards, the SPAS is asked to request the measurements it needs for the analysis of static performance. The application is then released and runs for n iterations of the progress loop while the requested measurements are collected.

Upon the completion of the last iteration in the burst, the default dynamic context specifying the temporal access window is positioned on the interval of iterations just executed. This access window is then transparently used to compute aggregated values of metrics over of measurement samples collected during the experiment. These are then evaluated by the SPAS resulting in a set of found static properties. They specify the type of the problem as well as its location and severity. PDAS queries them for the in-detail dynamics analysis of the found performance inefficiencies.

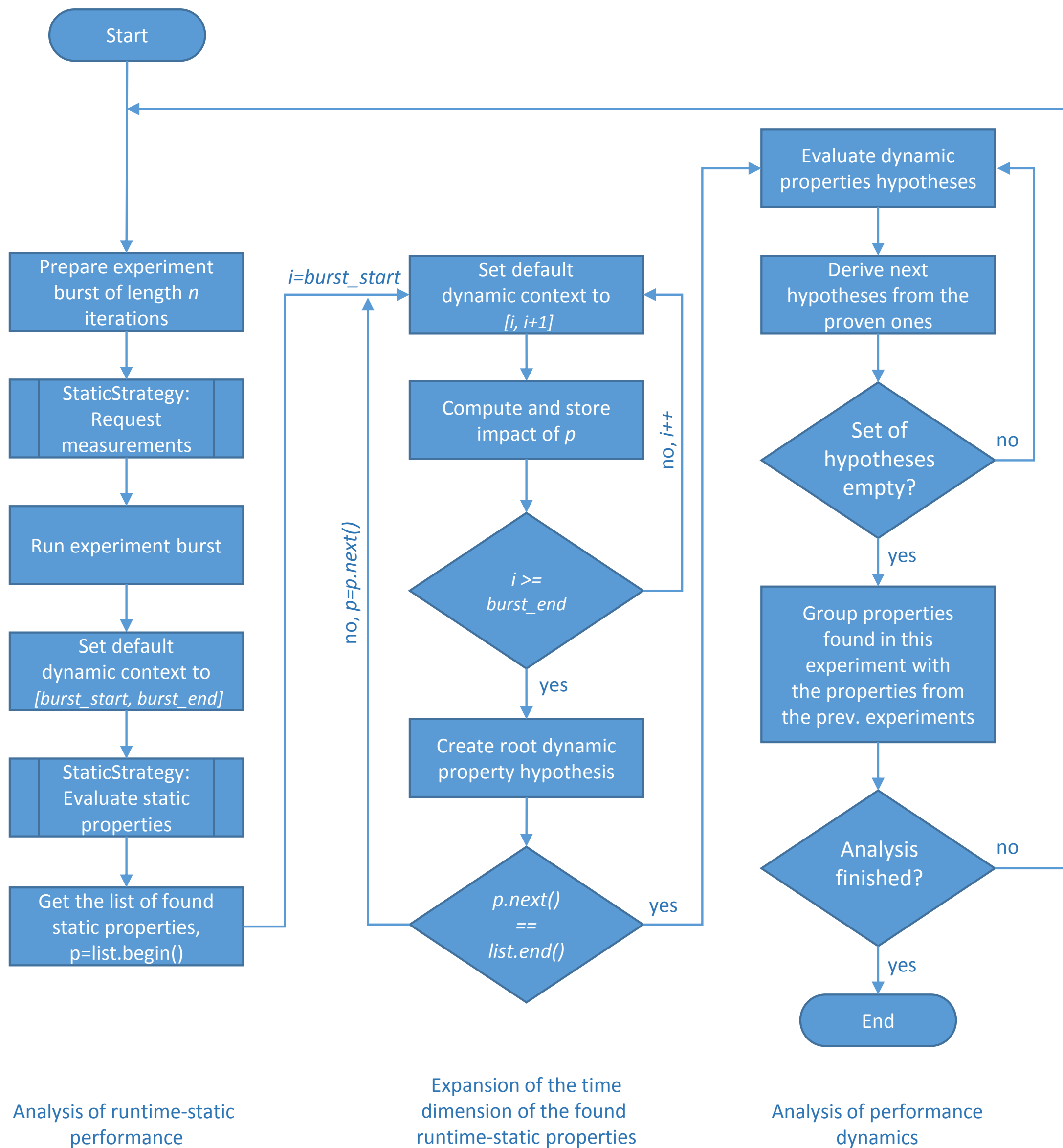


Figure 6.9: Algorithm of the Performance Dynamics Analysis Strategy

Expansion of the time dimension for the found runtime-static properties

Before evaluating dynamics of a detected performance problem, a time-series characterizing its temporal evolution has to be obtained. This is done in the second phase of the analysis, shown in the middle of the Figure 6.9, where the temporal dimension of the found performance inefficiencies is expanded. This is done by iteratively evaluating each found SP for each iteration, i, of the experiment burst. The result is a set of time-series

of severity values which quantify relative impact of the problem as follows:

$$Severity_i = \frac{Impact_i}{PhaseTime_i}, \tag{6.7}$$

where $Impact_i$ is the negative influence of the problem computed in time units lost and $PhaseTime_i$ is the total execution time of the phase region measured during iteration i. However, a time-series of relative values of severity might be misleading when the performance dynamics is of interest. Consider an example when the application has two performance problems each responsible for 50% of the execution time (an artificial example when the execution is composed of two phases both evaluated as an inefficiency). Now when we double the time lost due to both problems, the severity of each is still the same 50%. Therefore, what we need is a time-series of absolute severity values, i.e. impact, which we derive from the severity values using the formula above. Furthermore, for each time-series of SP's impact values we create the root hypothesis about performance dynamics which is evaluated in the next phase.

Analysis of performance dynamics

In the third phase of the analysis algorithm the hypotheses about degradations in the temporal evolution of the detected SPs are tested. During the evaluation the impact time-series derived above are processed by means of the signal processing algorithms presented in the beginning of this chapter. The hypotheses are then accepted or rejected by evaluating pre-defined inference rules against the resulting dynamic features.

When a hypothesis is accepted we say a performance dynamics property is found. In the next step these are refined by instantiating derived hypotheses which are specified by the hypotheses hierarchy. If the set of hypotheses is not empty, the next is evaluated and refined until the whole hierarchy is parsed.

As a result of the evaluation, a number of performance dynamics properties are found for the current burst of iterations. Following the online analysis scheme presented in the previous chapter we group the properties found in this burst with the properties found in the previous ones in order to prevent linear increase in the collected data as well as in the effort of interpretation of the analysis results. The grouping of the properties of the same type is performed along the time dimension according to a property-specific aggregation specification. After grouping is completed, merged properties are kept for the next experiment burst which is entered unless the analysis is finished. This can happen when either the application has terminated or the requested analysis duration is reached.

6.4.3 Adapted APART Property Specification Language

We adapt the APART property specification language [26],[32] to define performance dynamics properties.

A performance property specifies an aspect of the inefficient application behavior. Each property is defined within a context which can include a source code region, a process and a thread. In our extension one or more properties can serve as a context of another property. Additionally, we need to define the temporal context for the property evaluation which is specified by a dynamic context.

In order to simplify reading, each specification has a definition section where functions or constants used in the following specification section are given.

The actual property specification consists of the following features:

- **condition** - a rule to check the hypothesis about property existence expressed in functions and constants specified in the definition section
- **confidence** - an expression used to quantify the confidence in case the existence of the property cannot be proven, otherwise it is equal to 1.
- **severity** - an expression quantifying the value of negative impact due to the evaluated performance aspect. It is normalized so that the global ranking between multiple properties is possible. We require that the value is normalized to the execution time of the phase region aggregated over the given dynamic context.

We extend the list above with the **next** function which returns a set of derived hypotheses if the condition of the property is evaluated as true. This is used to define hierarchical relations between properties.

Furthermore, in order to specify the temporal extent of the detected dynamic property we extend the specification with the **extent** expression which returns the dynamic context of the found property. This might be equal to the input dynamic context which is used to define the evaluation domain, or it could be a sub-context when the property is localized within the initial domain.

6.4.4 Performance Dynamics Properties

In this section we present specifications of performance dynamics properties (DP). These formalize possible degradations in the dynamics of performance problems represented by SPs. In addition to the description of the problem its location and severity inherited from the SP, each DP specifies:

- a high-level and concise description of the temporal pattern native to the "mental" model of the user,

- temporal location,
- severity quantifying negative impact due to the degradation.

Significant Variability Property

The first property in the hierarchy of performance dynamics properties is the "Significant Variability" property which specification is given in the Figure 6.10. We use this to check whether the impact of a given SP is changing with the time.

The context for this property, provided as an input, is a SP and a dynamic context. The first specifies the problem being observed in the application and a static context where it was observed. We use the property id as a key to access the pre-computed time-series of impact values.

Instead of formalizing evaluation rules for dynamic properties in raw temporal performance data, we use dynamic features extracted from time-series of measurements by the signal processing algorithms described in the previous sections. We use energies of the DWT coefficients which quantify variability observed in the signal over all scales and temporal locations. Thanks to the orthonormality of the transform, we can compare it to the total energy of the signal to get a relative quantity which can be compared against a predefined threshold. This rationale is used to specify the property `condition`.

Unfortunately, we cannot use it to quantify the severity value which has to be normalized to the aggregated execution time of the phase region. Therefore, the severity of this property is always zero. However, it is not a problem since the goal of the property is to filter out SPs which do not show any dynamics with time and by this prevent the evaluation of further DPs.

If the condition of the property is true the *next* function returns hypotheses about degradation trends and/or degradation peaks in the dynamic behavior of the SP.

Since the significant variability was detected for the whole dynamic context provided as an input, the `extent` of the found property is identical.

In order to group two "Significant Variability" properties along the time dimension we provide an alternative specification of the property which requires two original properties as a context and the new dynamic context as an input. The specification is shown in Figure 6.11.

The condition in the definition section of the property specification verifies that only two dynamic properties belonging to the same static context can be grouped with each other.

The evaluation logic follows the original specification, except that the dynamic energy of the aggregated property is calculated as a sum of the dynamic energies of the both input properties. Analogously, the total signal energy is calculated. The extent of the merged property, if positively evaluated, equals to the extended dynamic context.

```
PROPERTY significantVariability(Property sp, DC dynamicContex)
{
 LET
     staticContext = sp.getStaticContext();
     metric = sp.id();
     dwt = pdb.getDWT(staticContext,dynamicContext,metric);
     dynamicEnergy = dwt->getDynEnergy();
     totalEnergy = dwt->getTotalEnergy();
 IN
     condition  : dynamicEnergy/totalEnergy > thresholdEnergy
     confidence : 1.0;
     severity   : 0;
     next       : DegradationPeaks(sp,dynamicContext),
                  DegradationTrends(sp,dynamicContext);
     extent     : dynamicContext
}
```

Figure 6.10: Specification of the “Significant Variability“ property

```
PROPERTY significantVariability(significantVariability p1,
significantVariability p2, DC dynamicContex)
{
 LET
    if(p1.staticContext!=p2.staticContext)
       group=false;
    dynamicEnergy = p1.dynamicEnergy + p2.dynamicEnergy;
    totalEnergy = p1.totalEnergy + p2.totalEnergy;
 IN
    condition  : group && dynamicEnergy/totalEnergy > thresholdEnergy
    confidence : 1.0;
    severity   : dynamicEnergy/totalEnergy;
    next       :
    extent     : dynamicContext

}
```

Figure 6.11: Grouping specification of two “Significant Variability“ properties

```
PROPERTY DegradationPeaks(Property sp, DC dynamicContex)
{
 LET
     staticContext = sp.getStaticContext();
     phaseContext = sp.getPhaseContext();
     metric = sp.id();
     ssf = pdb.getSSF(staticContext,dynamicContext,metric);
     peaks = SSF->searchSequence('AB',thresholdStability);
     accumTime = peaks[0].getSum() + ... +
                  peaks[peaks.size()-1].getSum();
     dwt = pdb.getDWT(staticContext,dynamicContext,metric);
     shortScaleEnergy = dwt->getShortScalesEnergy();
     totalEnergy = dwt->getTotalEnergy();
     phaseTime = pdb.getSum(phaseContext,dynamicContex,EXECUTION_TIME);
 IN
     condition  : accumTime/phaseTime > thresholdTime &&
                  shortScaleEnergy/totalEnergy > thresholdEnergy;
     confidence : 1.0;
     severity   : accumTime/phaseTime;
     next       : DegradationPeak(sp,dynamicContext,peaks[0]),
                  ,...,
                  ,DegradationPeak(sp,dynamicContext,peaks[peaks.size()-1]);
     extent     : dynamicContext
}
```

Figure 6.12: Specification of the "Degradation Peaks" property

Degradation Peaks Property

As it follows from the name, the property checks the presence of temporarily localized peaks in the dynamic behavior of a given SP during the time interval specified in a given dynamic context. The specification is provided in Figure 6.12.

Considering that the analyzed dynamic behavior reflects multiple factors which result in dynamic features of different magnitude, span and location, detection of relevant peaks is not easy. Consider the example signal shown in Figure 6.1. Formally speaking, every small bump in the signal value can be seen as peak. Instead, when manually evaluated only the peak around iteration 20 appears relevant. Using the SSF technique capable of extracting qualitative representation of signals as well as providing a quantitative measure for visual relevance of a feature we are able to give a concise formalized rule for evaluating such peaks in the temporal behavior of the studied performance problems.

First, we perform the SSF analysis of the time-series of impact values of the input SP. Then we traverse the resulting interval tree in the breadth-first fashion searching for

```
PROPERTY DegradationPeaks(DegradationPeaks p1, DegradationPeaks p2, DC
dynamicContext)
{
 LET
    if(p1.staticContext!=p2.staticContext)
       group=false;
    shortScaleEnergy = p1.shortScaleEnergy + p2.shortScaleEnergy;
    totalEnergy = p1.totalEnergy + p2.totalEnergy;
    peaks.insert(p1.peaks);
    peaks.insert(p2.peaks);
    accumTime = peaks[0].getSum() + ... +
                peaks[peaks.size()-1].getSum();
    phaseTime = pdb.getSUM(p1.phaseContext,dynamicContex,EXECUTION_TIME);
 IN
    condition  : group && accumTime/phaseTime > thresholdTime &&
                 shortScaleEnergy/totalEnergy > thresholdEnergy;
    confidence : 1.0;
    severity   : accumTime/phaseTime;
    next       :
    extent     : dynamicContext
}
```

Figure 6.13: Grouping specification of two "Degradation Peaks" properties

"AB" sequences of qualitative geometric descriptors which correspond to the shape of a peak. As mentioned, there are possibly dozens of peaks present in the signal. In order to filter out those which are not relevant we impose a minimum stability constraint, which corresponds to the visual "relevance" of a peak. For each peak we compute the accumulated impact value, which is a sum of the time-series samples belonging to the peak. Since it is required that the impact values are measured in time units, these are summed up to get the accumulated time measure and stored in the corresponding variable.

Additionally, to quantify the contribution of the short scale variability, of which the peaks are part of, we use Wavelet transform to calculate the ratio of the energies of short scales to the total energy of the signal. This value is used together with the accumulated time of all detected relevant peaks in order to evaluate the `condition`. The `severity` is then simply computed as the accumulated time normalized by the total execution time of the phase region.

If the property is evaluated as true, a set of hypotheses is returned by the `next` function which are instances of the "Degradation Pattern" property. These evaluate each detected peak individually.

Temporal grouping of two properties of type "Degradation Peaks" is specified in Figure 6.13. The evaluation logic follows the original property specification except that now the relevant peaks are obtained not from the signal (it is not anymore available for the properties from the last experiment), but simply by concatenating the arrays of peaks from both properties. Same holds for the energies values. Of course, only the properties within the same static context can be merged.

Degradation Trends Property

Gradual deterioration of a performance characteristic is an important dynamic degradation pattern. These are especially difficult to detect by other techniques. When visualized by value maps visual interpretation of trends is not easy due to low differences in the color intensity of the neighboring points. With the "Degradation Trends" property we formalize the degradations due to gradual increase in the impact values of the detected SPs. The specification is given in Figure 6.14.

The context of this property is defined by a SP and a dynamic context. The first argument defines the static context and the metric id allowing to identify the time-series of impact values to be searched for degradation trends.

Similar to the "Degradation Peaks" property we use SSF to obtain a hierarchical qualitative representation of the analyzed time-series. However, searching for all present trends in the signal over all scales, the approach taken in the previous property specification, most likely will result in multiple trends which are nested in each other. This happens, for example, when a trend is shortly interrupted by a spike. One can see this as one trend or two trends, depending on scale. Therefore, we rely on the stability value to select the

```
PROPERTY DegradationTrends(Property sp, DC dynamicContex)
{
 LET
     staticContext = sp.getStaticContext();
     phaseContext = sp.getPhaseContext();
     metric = sp.id();
     ssf = pdb.getSSF(staticContext,dynamicContext,metric);
     msl = ssf->getMSL();
     trends = msl.searchSequence('A','D','E');
     accumTime = trends[0].getSum() + ... +
                  trends[trends.size()-1].getSum();
     dwt = pdb.getDWT(staticContext,dynamicContext,metric);
     wideScaleEnergy = dwt->getWideScalesEnergy();
     totalEnergy = dwt->getTotalEnergy();
     phaseTime = pdb.getSUM(phaseContext,dynamicContex,EXECUTION_TIME);
 IN
     condition  : accumTime/phaseTime > thresholdTime &&
                  shortScaleEnergy/totalEnergy > thresholdEnergy;
     confidence : 1.0;
     severity   : accumTime/phaseTime;
     next       : DegradationTrend(sp,dynamicContext,trends[0]),
                  ,...,
                  ,DegradationTrend(sp,dynamicContext,trends[trends.size()-1]);
     extent     : dynamicContext
}
```

Figure 6.14: Specification of the "Degradation Trends" property

```
PROPERTY DegradationTrends(DegradationTrends p1, DegradationTrends p2, DC
dynamicContext)
{
 LET
    if(p1.staticContext!=p2.staticContext)
       group=false;
    wideScaleEnergy = p1.shortScaleEnergy + p2.shortScaleEnergy;
    totalEnergy = p1.totalEnergy + p2.totalEnergy;
    trends.insert(p1.trends);
    trends.insert(p2.trends);
    accumTime = trends[0].getSum() + ... +
                trends[trends.size()-1].getSum();
    phaseTime = pdb.getSUM(p1.phaseContext,dynamicContex,EXECUTION_TIME);
 IN
    condition  : group && accumTime/phaseTime > thresholdTime &&
                 wideScaleEnergy/totalEnergy > thresholdEnergy;
    confidence : 1.0;
    severity   : accumTime/phaseTime;
    next       :
    extent     : dynamicContext
}
```

Figure 6.15: Grouping specification of two "Degradation Trends" properties

scale which provides the most visually relevant representation, which is the maximum stability level of the interval tree.

After obtaining the MSL we search it for consequent sequences of qualitative descriptors representing rising shapes. These are the "A","D" and "E" primitives. Each detected sequence is then stored as one trend in the array `trends`.

Similarly to the "Degradation Peaks" property, we use accumulated time together with wavelet energies to evaluate `condition` and `severity` expressions. Since with this property we evaluate the presence of trends, we use wide-scale Wavelet coefficients to quantify variability caused due to "slow" changes in the signal.

If the property is positively evaluated the `next` function returns the "Degradation Pattern" property which evaluates each found trend individually.

The grouping of two "Degradation Trends" properties is shown in Figure 6.15 and follows the same logic as the original specification, except that sequences of trends are not extracted from the signal, but directly obtained from two properties being merged. Also the `condition` and the `severity` values are re-evaluated against the extended dynamic context.

Degradation Pattern Property

This property refines collectively evaluated degradation patterns such as a trend or a peak by evaluating them individually. The necessity for this property in the presence of the collective ones is that it checks the significance of a single pattern. It can be the case that a number of patterns together result in the parent property to be found, however, no individual pattern is worth to be reported. The property specification is given in Figure 6.16a.

In addition to a SP and a dynamic context, a qualitative sequence, representing the pattern to be evaluated, is provided as an input. Similarly to the previous specifications, the evaluation is based on the accumulated impact value of the pattern which is normalized to the total time of the phase region. The `next` function return an empty set, since we don't further refine a single pattern.

The temporal location of the degradation pattern is a sub-interval of the dynamic context for which the property was evaluated. Therefore the `extent` function returns a dynamic context describing the precise temporal location of the pattern.

Temporal grouping of the "Degradation Pattern" property is different since it is not meaningful to try to merge two independent patterns into one property of this type. However, when a new burst of iterations is measured in the next experiment, the `condition` and `severity` values of the property have to be re-evaluated for the extended dynamic context. This is done using the specification given in Figure 6.16b.

6.5 Summary

The knowledge about performance dynamics is crucial in order to recognize optimization potentials. Direct manual interpretation of raw temporal performance data is overwhelming when large, long-running applications are studied. The reason is that the effort is proportional to the length and the number of measurement time-series which are a cross-product of application regions, processes and metrics.

While manually analyzing the temporal evolution of a performance characteristic, typically presented as a 1D plot, 2D value map or a 3D plot, relevant degradation patterns and their temporal locations are of interest. In this chapter we have presented a technique for the automatic detection of such patterns.

The technique consists of an online analysis procedure, signal processing algorithms and a set of inference rules producing high-level knowledge about performance dynamics.

Following the online analysis procedure, the performance inefficiencies are identified first in the global context by answering the questions "what" the problem is and "where" it is located. Then the impact of the detected problems is evaluated along the temporal dimension.

```
PROPERTY [Peak/Trend]SingleDegradation(Property sp, DC dynamicContex,
qualSequence sequence)
{
 LET
     phaseContext = sp.getPhaseContext();
     accumTime = sequence->getSum();
     location  = sequence->getDynamicContext();
     phaseTime = pdb.getSUM(phaseContext,dynamicContex,EXECUTION_TIME);
 IN
     condition  : accumTime/phaseTime > thresholdTime;
     confidence : 1.0;
     severity   : accumTime/phaseTime;
     next       :
     extent     : location
}
```

(a)

```
PROPERTY [Peak/Trend]SingleDegradation(DegradationPeak p1, DC dynamicContex)
{
 LET
     phaseContext = p1->getSP->getPhaseContext();
     accumTime = p1->getSequence->getSum();
     location  = sequence->getDynamicContext();
     phaseTime = pdb.getSUM(phaseContext,dynamicContex,EXECUTION_TIME);
 IN
     condition  : accumTime/phaseTime > thresholdTime;
     confidence : 1.0;
     severity   : accumTime/phaseTime;
     next       :
     extent     : location;
}
```

(b)

Figure 6.16: (a) Specification of the "Degradation Pattern" property, (b) re-evaluation of the "Degradation Pattern" property for the extended dynamic context

The resulting time-series are passed to the signal processing algorithms. These provide quantification of variability both along the time and the scale dimension; multi-scale representation; qualitative representation as well as quantification of the visual "relevance" of patterns. The outcomes form an intermediate representation offering a compact representation capturing both quantitative and qualitative features of the signal.

The high-level knowledge about performance degradations is then derived using the inference rules represented by performance dynamics properties. These formalize degradations such as significant peaks or trends in the severity of the detected performance inefficiencies.

Chapter 7

Evaluation

In this chapter we demonstrate the online collection and automatic performance dynamics techniques presented in two previous chapters. For the evaluation we select four real-world applications which we analyze, first manually and then automatically, in the four following case studies:

- **CX3D** - a simple performance dynamics case study intended to explain techniques proposed in this thesis.
- **SPEC MPI2007 129.tera_tf** - in the presence of a complex performance dynamics behavior we demonstrate how the automatic analysis algorithm proposed in this study provides insights beyond the ones obtained by manual analysis.
- **PEPC** - an application running for 2000 iterations which is used to evaluate our technique for performance dynamics analysis of long-running applications.
- **INDEED** - production metal-forming simulation code for which a knowledge about performance dynamics obtained using our technique is used to improve overall performance.

7.1 Experimental Setup

For the performance analysis experiments "SuperMUC" [9] - the IBM System x iDataPlex (thin nodes) supercomputer maintained by the Leibniz Supercomputing Centre was used. With more than 155.000 cores and a peak performance of 3 Petaflop/s (= 10^{15} Floating Point Operations per second) in June 2012 SuperMUC is one of the fastest supercomputers in the world.

The system consists of 18 thin node islands and one fat node island interconnected via Infiniband network. Thin nodes were used to run the experiments. Each node consists of

two Intel Sandy Bridge-EP processes with 16 cores in total.

Intel compiler v14.0 together with IBM's Parallel Environment (IBM MPI) were used to compile applications. Score-P v1.2.3 and Periscope 1.6b were used to respectively instrument, monitor and analyze performance dynamics of the test applications.

7.2 CX3D - Czochralsky Crystal Growth Simulation

The goal of this study is to have a "look under the hood" of the presented analysis techniques. Throughout the study we will be referencing already presented intermediate results used as an example in the previous chapter.

7.2.1 Automatic Analysis of Performance Dynamics with Periscope

Before starting the analysis, the application has to be instrumented. This is done using the user instrumentation macros of Score-P. The code fragment below shows applied instrumentation:

```
1       #include "SCOREP_User.inc"
2       PROGRAM CX3D
...
36      SCOREP_USER_REGION_DEFINE(R1)
...
  C     - START OF THE TIME LOOP -
61 9999 IT=IT+1
62      SCOREP_USER_OA_PHASE_BEGIN(R1,"OP",SCOREP_USER_REGION_TYPE_COMMON)
63      T=T+DT
64      CALL CURR
65      CALL VELO(IER)
...
93      SCOREP_USER_OA_PHASE_END(R1)
94      IF( T+EPS .LE. TIME ) GOTO 9999
```

On the first line Score-P user instrumentation header is included. Then on line 36 a user instrumentation region handle is declared. It is used in macros marking the beginning and the end of the Online Access Phase region on lines 62 and 93. The latter has to be an iteratively executed part of the application with a potential for global synchronization at the places where the macros are inserted. Iterations of the phase region define the temporal dimension of the applied analysis. Therefore, it is important that the phase is placed within the body of the application progress loop.

Afterwards the application can be compiled and instrumented using the following command:

```
scorep --online-access --user
```

The analysis with Periscope is started by issuing the following command:

```
psc_frontend --apprun=./cx --strategy=PerfDyn-SingleCore --duration=128
--phase=''OP''
```

By default Periscope starts the application with 1 process and 1 OpenMP thread, which is equivalent to a sequential execution. Without going into details we describe below the main steps of the analysis process.

Starting Periscope agents and application process. During this step analysis agents (and high-level reduction agents if needed) are started. Also the application is automatically started. A registry service is used to identify peers for the distributed agents. After being started the application runs until the beginning of the online access phase and is suspended there.

Analysis initialization. After the analysis hierarchy is up and running, the analysis agent initializes requested analysis strategy, which is the Performance Dynamics Analysis Strategy (PDAS). As a post-fix to the strategy name in the start command we specified that Single-Core Performance Analysis Strategy (SCPAS) should be used to identify performance properties which are then studied for temporal dynamics. From the implementation point of view the PDAS is a meta-strategy which drives the SCPAS.

Requesting measurements. In this steps the PDAS tells the SCPAS to issue measurement requests. In current setup these are requests to measure execution time of the phase region. Afterwards, PDAS requests the Data Provider (see Chapter 5.5) of Periscope to perform a burst of 64 phase region iterations. When the requests are submitted to the monitoring library the application is released for execution.

Monitoring. Following the Online Dynamic Profile Analysis scheme (see Section 5), collected profile data is transmitted at the end of each iteration to the analysis agents and stored in the Performance Data Base. Therefore, regardless of how long the analysis is performed the amount of data buffered on the application side is not growing with time.

Analysis of single core performance. After the configured series of iterations is completed, buffered performance data is first analyzed for the presence of single core performance properties. The properties, in current example only the “Hot spot” properties, are evaluated against the performance data reduced over the buffered series of profile samples. As a result, one property was found which is presented in the first row in Table 7.1.

Table 7.1: Properties detected in iterations interval [1-64] of the CX3D progress loop.

Property description	Iterations	Region	Process	Severity
Hot spot of the application	[1,64]	`velo`	0	86%
Significant variability	[1,64]	`velo`	0	-
Degradation trends	[1,64]	`velo`	0	35%
Degradation peaks	[1,64]	`velo`	0	12%
Degradation trend	[1,21]	`velo`	0	35%
Degradation peak	[6,21]	`velo`	0	12%

Analysis of performance dynamics. The runtime-static performance property found in the previous step is now analyzed for dynamic properties. Following the analysis algorithm proposed in Section 6.4.2, the severity of the detected property is re-evaluated in each iteration of the phase region. Resulting series of severity values is then used to evaluate performance dynamics of the property. The first performance dynamics property to be evaluated is called "Significant Variability". By means of Discrete Wavelet Transform it checks whether the temporal dynamics of the analyzed performance characteristic is worth to be further analyzed. In our case, as it can be seen from the Table 7.1, the property is positively evaluated. However, since the energy of Wavelet coefficients used to evaluate the property cannot be normalized to time, the severity of this property cannot be reported.

The found property is refined and two new properties are evaluated: "Degradation Trends" and "Degradation Peaks". These properties require SSF and use maximum stability level of the resulting qualitative multi-scale representation to check for increasing trends and peaks respectively. The maximum stability level of the test signal was already presented in Figure 6.4 in Section 6.3. As it can be seen in Table 7.1, both properties were found meaning that the temporal development of the analyzed performance characteristic is impacted by degradation trends and peaks. The first accounts for 35% of the total phase execution time in iterations interval [1-64]; the latter accounts for 12% of the total phase execution time over the same interval.

In order to analyze individual trends and peaks the two found properties are refined into multiple "Degradation Pattern" properties, one per each found pattern. The result of evaluation are two last rows in the Table 7.1. It appears that there is only one trend localized in iterations interval [1,21] and one peak localized in iterations interval [6,21]. Looking at the original signal (Figure 6.1) one can clearly see both temporal features.

Analysis of the next chunk of iterations. Since there are no more properties to be evaluated, the analysis of the first chunk of 64 iterations is complete. According to the online analysis scheme the next burst of 64 phase region iterations are requested and, when completed, analyzed repeating the steps presented above. The properties found in the second half of the execution are presented in Table 7.2. As one can see, there is only one runtime-static property detected in the second half of the execution, i.e. iterations

Table 7.2: Properties detected in iterations interval [65-128] of the CX3D progress loop.

Property description	Iterations	Region	Process	Severity
Hot spot of the application	[65,128]	`velo`	0	85%

Table 7.3: Final set of properties reported for the CX3D application.

Property description	Iterations	Region	Process	Severity
Hot spot of the application	[1,128]	`velo`	0	86%
Degradation trend	[1,21]	`velo`	0	27%
Degradation peak	[6,21]	`velo`	0	6%

65-128, and no performance dynamics properties. Indeed, comparison of the result with the plot in Figure 6.1 shows that the temporal behavior in the second half of the signal does not experience severe dynamics.

Grouping of properties along the time dimension. The properties detected in the first and second chunks of iterations have to be now merged along the time dimensions. In particular the severity values have to be recalculated against the extended dynamic context which now includes both chunks. Merging is performed by means of corresponding grouping property specifications presented in Section 6.4.4. In the final list we do not include the properties which were successfully refined into more specific properties.

The final report shown in Table 7.3 contains only three properties. The first one states the type and the severity of the detected performance inefficiency. The next two provide valuable insights into the dynamics of the problem specified by the first property. The second row in the table highlights a rapidly increasing trend in the absolute severity values, i.e. execution time of the `velo` subroutine, in iterations interval [1-21]. It accounts for 27% of the phase region total execution time. In other words, should the trend be prevented, the total execution time of the phase region can be decreased by 27%. The last property reports a peak in iterations interval [6-21] accounting for 6% of the total phase region execution time.

7.3 SPEC MPI2007 129.tera_tf

129.tera_tf [1][49] is a 3D eulerian hydrodynamics application from the benchmarking suite SPEC MPI2007. Although, the suite consists of a number of applications we chose the 129.tera_tf due to its complex performance dynamics.

The application is a relatively small FORTRAN 90 application using MPI for parallelization. The domain is a cube with N cells in each dimension. It is split into blocks, where N has to be divisible by the block size in each dimension. One block corresponds to an MPI rank. The MPI implementation mainly uses point-to-point non-blocking communications

and collective reductions.

For our analysis we set the number of cells in each dimension to 120. We use 32 MPI processes which split the global domain into 4 blocks along the first two dimension and 2 blocks along the third one. The application is run for the first 128 iterations.

We use Score-P to instrument only the MPI call sites. Then Periscope is run in debugging mode which allows to dump raw temporal performance data in files which are then used as reference for the automatic analysis. In the second experiment the application is analyzed using Periscope Performance Dynamics Analysis Strategy.

7.3.1 Visual Analysis of Raw Temporal Performance Data

129.tera_tf is a good example of how a relatively small application with few MPI calls can be a hard subject for a performance dynamics study when running with only 32 processes. By measuring only MPI-related metrics the number of resulting time-series is already 128. Obviously, it is not anymore possible to plot and visually investigate each time-series individually. Therefore, we use value maps to plot time-series of measurements collected for the same MPI call site together. The y-axis depicts MPI ranks and the x-axis is the iteration of the progress-loop. The color-coded value represents the value of the metric in seconds.

We start the analysis by looking at the execution time of the phase region, i.e. the body of the progress loop, over the first 128 iterations of the progress loop, see Figure 7.1. The first anomaly, immediately cached by the eye, are four vertical high-intensity lines near iterations 10, 55, 70, 120, as well as several others with lower intensities. These are indicates of spontaneous peaks in the execution time of the phase region. Typically such spikes are the result of IO activities such as checkpointing.

Further investigating the value map one can see, although, looking very closely, a gradual increase trend in the execution time. However, it is very difficult to recognize its temporal limits since these are distorted by the short-scale variability in the second half of the execution.

The conclusion is that the phase region execution time is rising and has four prominent spikes near iterations mentioned above.

Point-to-point communication is represented by `MPI_Wait`. Figure 7.2 shows the evolution of the time spent waiting for the communications to complete over 128 iterations.

Dynamics of this metric is more complex than of the previous one. First, the dynamics differs from one process to another. Taking into account the semantics of `MPI_Wait`, this could be expected since the waiting time on one process is caused by a delay on the peer process, which, arriving later, does not have to wait itself. Another observation is that there is a short-scale variability from one iteration to another on a subset of processes. Finally, one can see that for a sub-set of processes the waiting time rises by the end of the

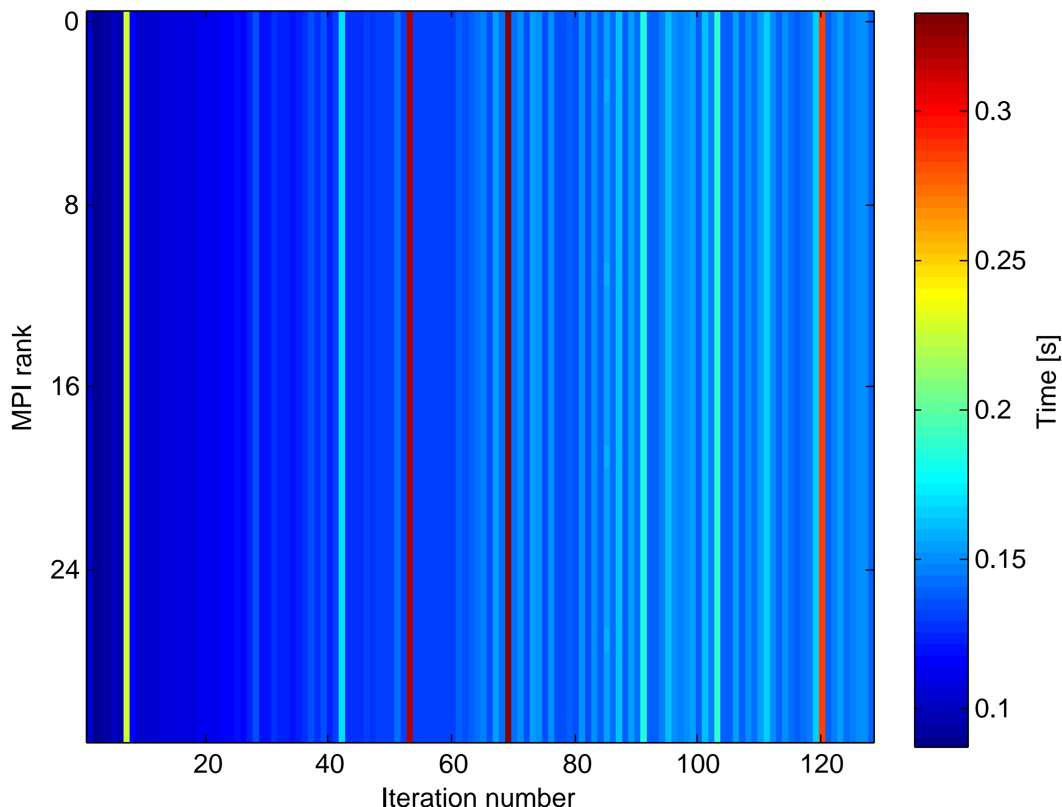


Figure 7.1: Value map of the phase region execution time over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis).

execution, however, it is very difficult to identify exact processes as well as the specific iterations interval where the trends manifest themselves.

Figure 7.3 presents the evolution of the collective communication time, measured for the `MPI_Allreduce` call, over the iterations of the progress loop. Here the pattern caught by the eye are the four vertical lines of high intensity familiar from the analysis of the phase execution time.

In this example we once again hit the limits of the visual analysis of performance dynamics. One has to look very precisely to recognize that the four lines are present on all processes except process 0. This means that the peaks on processes 1-31 in the execution time of the `MPI_Allreduce`, which also serves as a global synchronization, is the waiting time for process 0, which is apparently late. If we miss that process 0 is not affected by the pattern, we could have concluded that the reason for the increased collective communication time around iterations 10, 55, 70, 120 is due to communication overhead and not a synchronization issue.

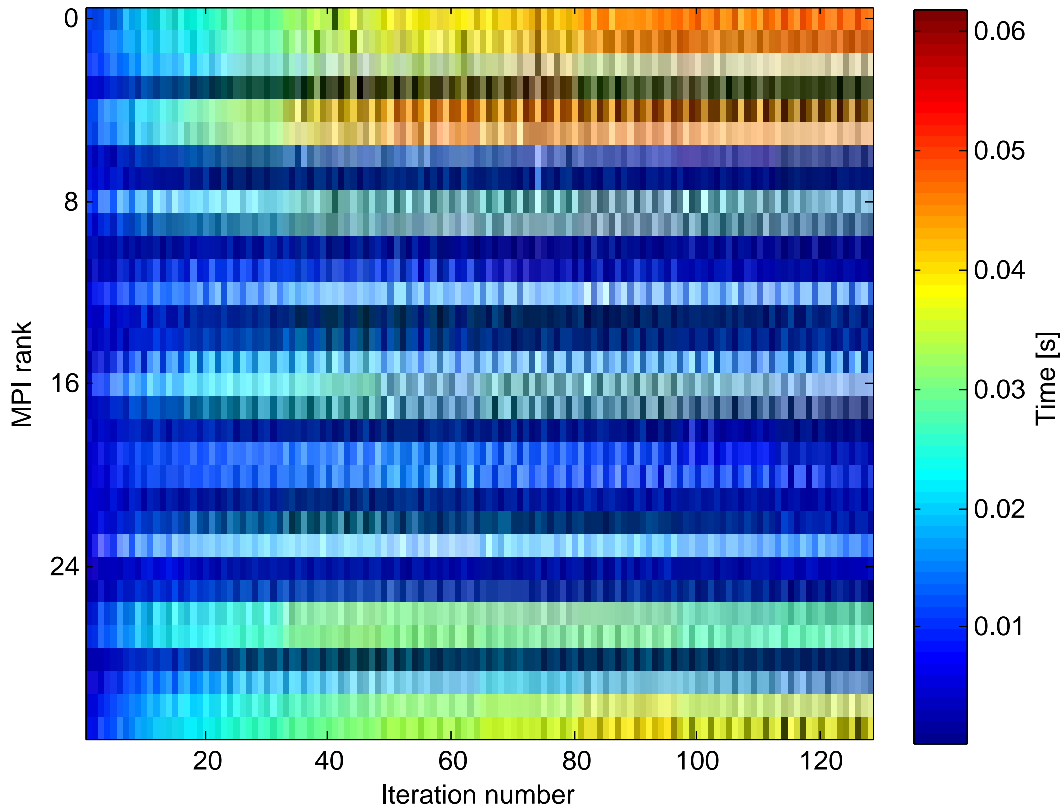


Figure 7.2: Value map of the communication time in the `MPI_Wait` call over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis).

7.3.2 Automatic Analysis of Performance Dynamics with Periscope

In this section we present performance properties automatically detected by Periscope. We run Periscope with the severity threshold equal to 10%. This means that only those properties are reported for which the severity, i.e. execution time wasted due to the reported problem in percentage to the total time, is greater than 10%.

Table 7.4 shows the properties reported by Periscope. The first three properties indicate three MPI-related performance issues. These indicate the type of the problem and where it was found, i.e. region and process.

The first two properties are two clusters of the same property type found in the same `MPI_Wait` call-site region. They are reported for two different sets of processes, since corresponding severity values significantly differ between two sets. One can also see that the two sets do not include all 32 processes, meaning that for some processes the problem was not detected. Comparing the properties with the corresponding value map (Figure

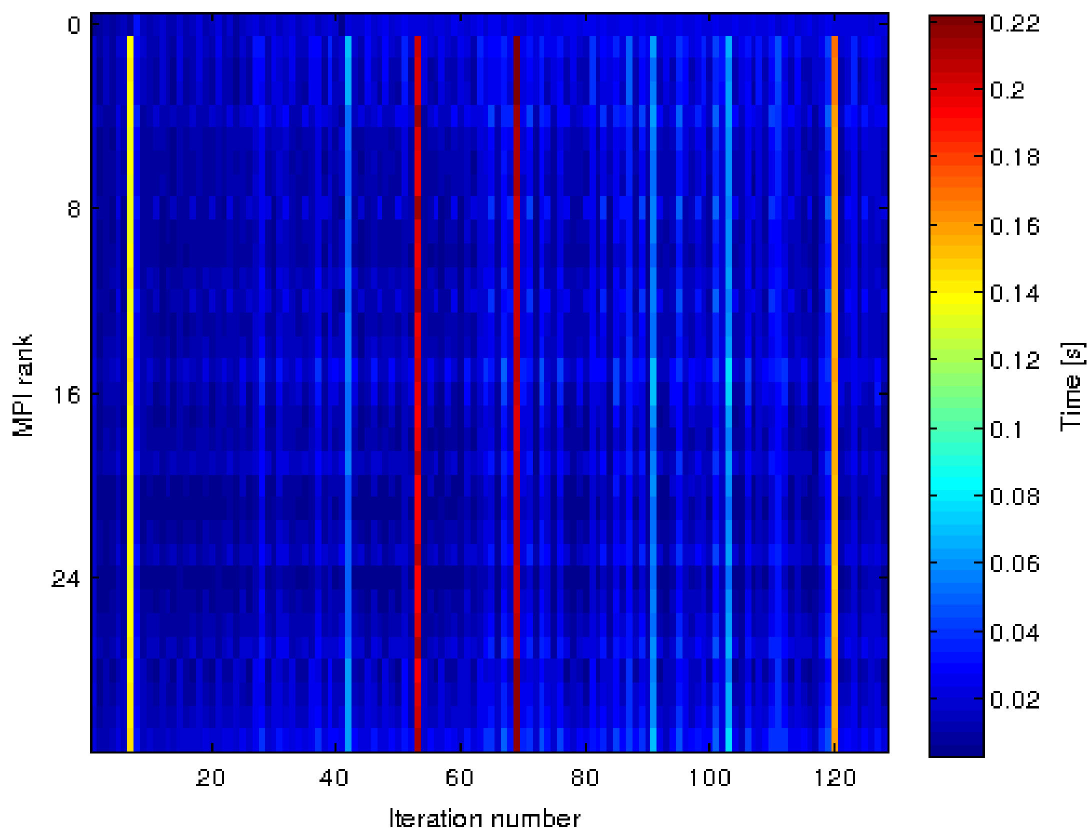


Figure 7.3: Value map of the communication time in the `MPI_Allreduce` call over the first 128 iterations of the progress loop (x-axis) and 32 MPI ranks (y-axis).

7.2) one can see that the high severity properties correspond two high-intensity rows. The third property report an excessive communication in the `MPI_Allreduce` call which is uniform over all processes.

Although being already an important piece of information, the properties do not say anything about how and when did the problem appear. This information can be obtained easily from the performance dynamics properties listed below. The first four of them list four degradation trends in the severity of the excessive communication time in `MPI_Wait`. The semantics of the properties explicitly tells that the problem is getting worth towards the end of the run, the conclusion we also came to during the manual analysis. In addition to that, Periscope provides further insights by reporting precise temporal span of each trend, the processes where it was observed, and a severity value which quantifies the amount of cumulative process-local execution time saved, should the trend be prevented.

The last property reports degradation peaks in the excessive communication time of

Table 7.4: Properties reported for the 129.tera_tf application

Property description	Region	Process	Severity
Excessive comm. time	`MPI_Wait`	0,1,2,4,5, 8,16,27,30, 31	26%
Excessive comm. time	`MPI_Wait`	9,12,15,17, 20,23,26,29	15%
Excessive comm. time	`MPI_Allreduce`	0-31	16%
Degradation trend in excessive comm. time in iterations [1,92]	`MPI_Wait`	4,27,30	17%
Degradation trend in excessive comm. time in iterations [1,128]	`MPI_Wait`	0,1,2,31	16%
Degradation trend in excessive comm. time in iterations [1,72]	`MPI_Wait`	5,8,12,16	15%
Degradation trend in excessive comm. time in iterations [1,51]	`MPI_Wait`	23,26,29	11%
Degradation peaks in excessive comm. time in iterations [5,7], [51,57], [67,69], [118,120]	`MPI_Allreduce`	1-31	4%

`MPI_Allreduce`. Although the severity of the property is below the threshold of 10%, we decided to include it in the final report for two reasons. First, it shows that the performance dynamics analysis is capable of detecting, classifying and quantifying such peaks in temporal performance data. The second reason is to stress that despite the four peaks are clearly visible and stand-out in the value map, their importance in terms of the time lost is low. Being able to easily rank performance issues according to their potential for performance gain, should they be optimized out, allows efficient guidance of the tuning process.

7.4 PEPC

PEPC application was already introduced in Section 4.4.2, so we skip it here.

For the analysis only MPI call-sites are instrumented using Score-P. We use the body of the time-stamp loop in `pepce.f90` file as a phase for the dynamic phase profiling. `medium.para` is used as a configuration file which is modified to run the application for 2048 time stamps, i.e. iterations of the progress loop.

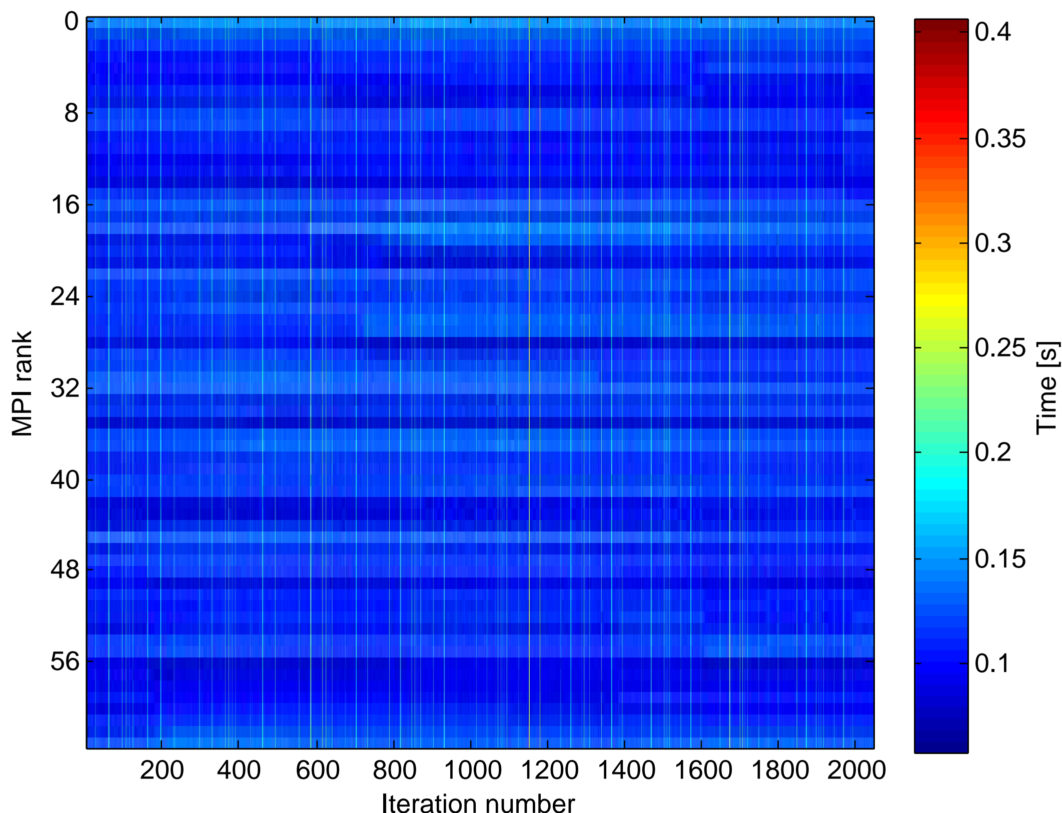


Figure 7.4: Value map of the communication time in `MPI_Allgather` call over the first 2048 iterations of the progress loop (x-axis) and 64 MPI ranks (y-axis).

7.4.1 Visual Analysis of Raw Temporal Performance Data

By instrumenting MPI calls and consequent collection of dynamic profiles results in 1280 time-series with 2560000 samples of MPI performance related metrics. However, by filtering out time-series collected for the MPI functions, which overall contribution to the performance is negligible, we are left with only two: `MPI_Alltoall` and `MPI_Allgather`. This shows the importance of first evaluating the significance of the problem for the overall execution and then investigating performance dynamics of the most severe inefficiencies. Like in the previous example we use value maps to visually investigate performance dynamics properties.

Figure 7.4 shows the temporal evolution of the `MPI_Allgather` communication time over the 2048 iterations of the progress loop recorded for all 64 processes. The most prominent temporal patterns are vertical, presumably one-iteration-wide, high-intensity lines. These indicate spikes in the communication time observed in the corresponding iterations. Since they are present in all processes, one can conclude that the reason is not a process being

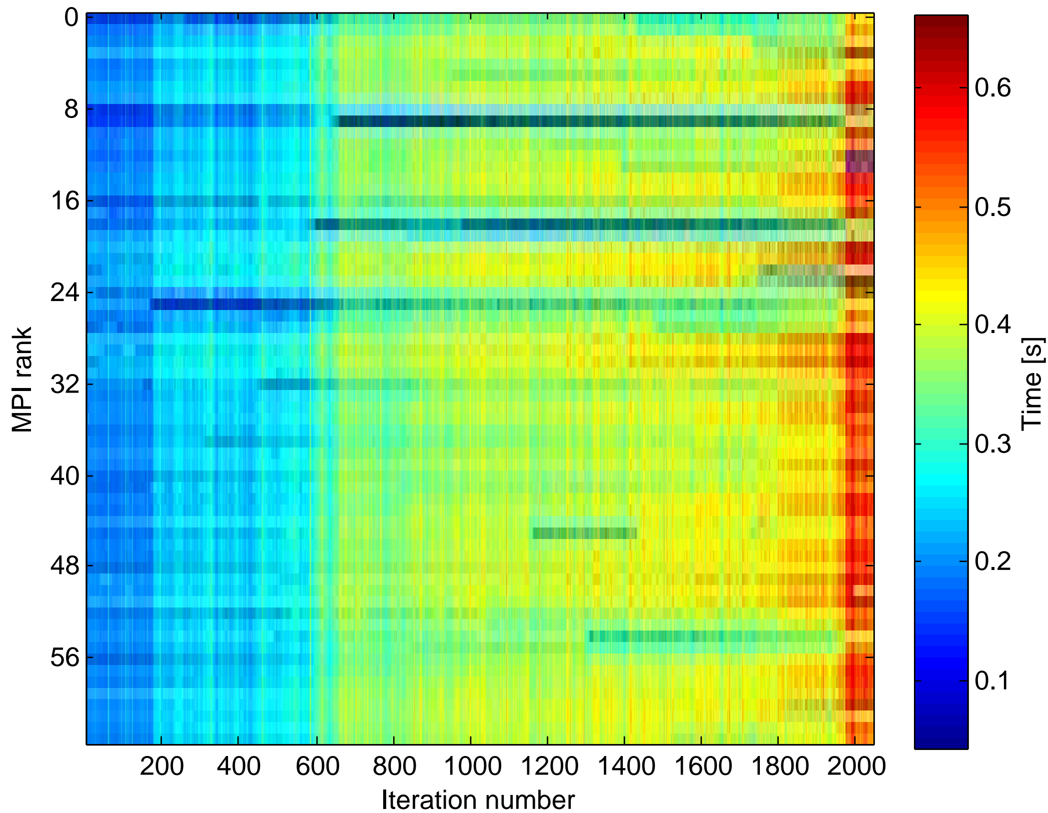


Figure 7.5: Value map of the communication time in the `MPI_Alltoall` call over the first 2048 iterations of the progress loop (x-axis) and 64 MPI ranks (y-axis).

late, but rather increased communication overhead. Another observation is that the process-local baseline is almost the same for all processes, however, there are variations across processes.

Temporal evolution of the `MPI_Alltoall` communication time is shown in Figure 7.5. The performance dynamics picture in this case appears to be more complex. First, one can clearly see that the value of the metric is increasing for almost all processes towards the end of the execution. However, it is difficult to grasp from the changes in the intensity when and how the degradation trend manifests itself. Furthermore, one can mention short-scale variability, although, low amplitude for all processes. Finally, processes 9, 18, and 25 show different patterns with almost constant low-magnitude values. This could be an indication of the synchronization issue, where the mentioned processes are late and the others, since `MPI_Alltoall` is a synchronization point, have to wait.

7.4.2 Automatic Analysis of Performance Dynamics with Periscope

In this section we present performance properties automatically detected by Periscope. In case of PEPC we lowered the severity threshold to 5% in order to keep performance dynamics properties. Furthermore, in order to reduce the amount of data buffered in the analysis agents we configured Periscope to perform online analysis in chunks of 1024 samples. Following the Performance Dynamics Analysis Strategy presented in the Section 6.4, each chunk is first automatically analyzed for the presence of performance dynamics properties. Resulting properties are then grouped along the time dimension. Table 7.5 presents properties found in the two analysis chunks.

The first three properties indicate two MPI inefficiencies of PEPC: excessive communication time in `MPI_Alltoall` and `MPI_Allgather`. The severity of the first issue is different across the processes. The first and larger group of processes shows average severity values of 15%. The second group consisting of processes 0, 9, 18 and 25 has severity of 9.6%. Comparing this result with the corresponding value map (Figure 7.5) in the previous section one can see that these match the low-intensity color stripes also identified in the manual analysis. The severity of the inefficiency detected in `MPI_Allgather` is uniform across processes and equals to 5%.

The next entries in the table list detected performance dynamics properties providing additional insights into temporal degradations detected in the temporal development of the inefficiencies presented above. The first property explicitly represents a degradation trend in the excessive communication time in `MPI_Alltoall` in the iterations interval [0-2048] for the majority of the application processes. This matches the observation made during the manual analysis. In addition to this knowledge the property reports the severity, which is a normalized cumulative communication time caused by the trend.

The next property reports two degradation trends in intervals [1-1028] and [1650-2048] for processes 1, 2, 6, 8, 20, 28. Apparently what was assumed a monotonous increase in communication time during the manual analysis does not hold for all processes. Instead, temporal degradation of the metric is localized in two intervals for the mentioned processes. Similarly a number of other processes deviate from the common pattern as depicted by the following three properties. Processes 26, 32 and 37 show increasing trend in communication time in interval [522-2048]; process 45 in intervals [1-1028] and [1200-2048]; and processes 22 only in [1-1028]. All mentioned performance dynamics properties report similar severity. By looking closely to the value map one can recognize that, indeed, the mentioned processes stand out from the common pattern. Although, the severity of the excessive communication time property in `MPI_Alltoall` for these processes was similar to the one reported for the majority of the processes, the dynamics of the inefficiency still differs. Such insights are often overlooked during manual analysis. Using our technique we were able to automatically extract this knowledge.

Finally, the last performance dynamics properties report degradation patterns observed

Table 7.5: Properties reported for the PEPC application

Property description	Region	Process	Severity
Excessive comm. time	MPI_Alltoall	0,9,18,25	9.6%
Excessive comm. time	MPI_Alltoall	1-8,10-17, 19-24,26-63	15%
Excessive comm. time	MPI_Allgather	0-63	5%
Degradation trend in excessive comm. time in iterations [1-2048]	MPI_Alltoall	3-5,7,10-12,14-17, 19,21,23,24,27,29-31, 33-36,38-44,46-53,55-63	6.7%
Degradation trend in excessive comm. time in iterations [1-1028], [1650-2048]	MPI_Alltoall	1,2,6, 8,20,28	6.7%
Degradation trend in excessive comm. time in iterations [522-2048]	MPI_Alltoall	26,32,37	6%
Degradation trend in excessive comm. time in iterations [1-1029], [1200-2048]	MPI_Alltoall	45	5.5%
Degradation trend in excessive comm. time in iterations [1-1028]	MPI_Alltoall	22	5.2%
Degradation trend in excessive comm. time in iterations [422-2048]	MPI_Alltoall	0,25	4%
Degradation trend in excessive comm. time in iterations [1-1200], [1400-2048]	MPI_Alltoall	54	4%
Degradation trend in excessive comm. time in iterations [1-612], [780-2048]	MPI_Alltoall	9	2%
Degradation trend in excessive comm. time in iterations [1,541], [1170-2048]	MPI_Alltoall	18	1.4%

on the processes 0, 9, 18, 25, which were previously identified as outliers. Although at different time locations and with significantly lower severity, these processes still show degradation trends towards the end of the iteration domain.

Apparently, there are no performance dynamics properties reported for the `All_Gather` call. The narrow one-iteration wide spikes in the metric value were disregarded due to negligible impact on the overall performance. In this case by reporting an empty set of performance dynamics properties, Periscope indicates that the temporal dimension is not relevant for the further analysis.

7.5 INDEED

INDEED [4] is a commercial sheet metal forming simulation software developed by GNS mbH. It is based on the incremental finite element method where the balance between inner and outer loads is determined in each incremental load step of the simulation.

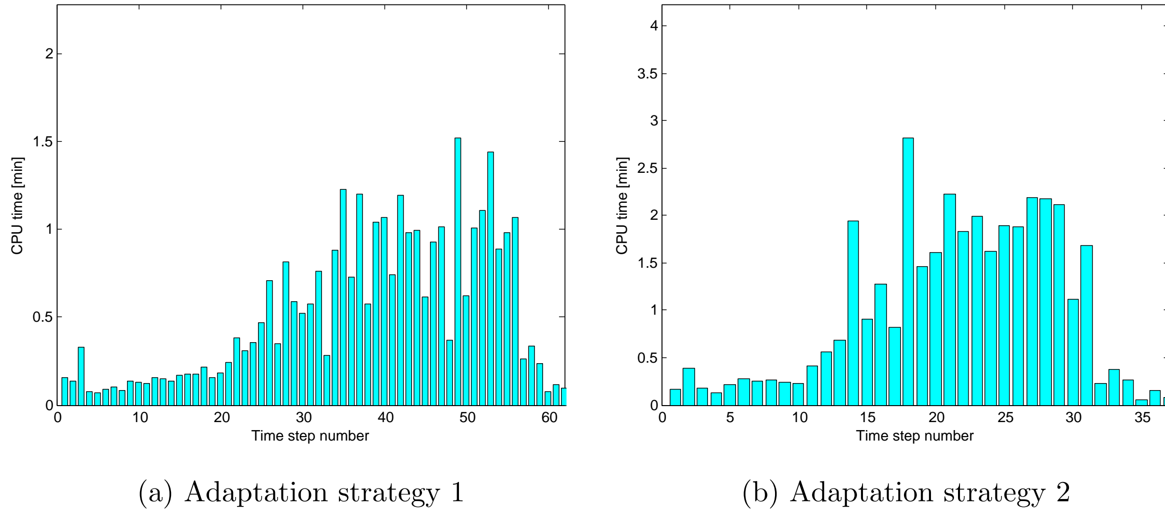
The major steps in the simulation algorithm are:

1. Assembly of stiffness matrices
2. Solution of stiffness systems
3. Recovery phase (computation of forces to see if equilibrium has been reached)
4. Mesh refinement
5. Analysis of contact between tools and workpiece

Further feature of INDEED is an adaptation of the time step width. This plays a crucial role for the performance of the application by influencing the number of iterations needed to reach the equilibrium in step 3 of the simulation algorithm presented above. The goal of the adaptation is to find the optimal trade-off between the time step width, and thus the number of time steps, on one side and the CPU time of the time steps, which in its turn is determined by the convergence of the recovery phase, on the other side. In this study we perform a performance dynamics study of the INDEED code with two time step adaptation strategies.

7.5.1 Experimental data

Unfortunately, the source codes of the INDEED version used in this study were not available for the direct experimentation. Therefore, the study is performed based on the experimental performance data provided by the leading developer of the INDEED code, Dr. Diethelm. We gratefully acknowledge the support provided by Dr. Diethelm.



(a) Adaptation strategy 1

(b) Adaptation strategy 2

Figure 7.6: CPU time of the time step iterations

Experimental data include CPU time records and time step width collected for the iterations of the progress loop. The application was run with one process and identical input data. Different time step adaptation strategies were monitored in order to find the one allowing the shortest execution time. In our study we use experimental data collected for the two following strategies:

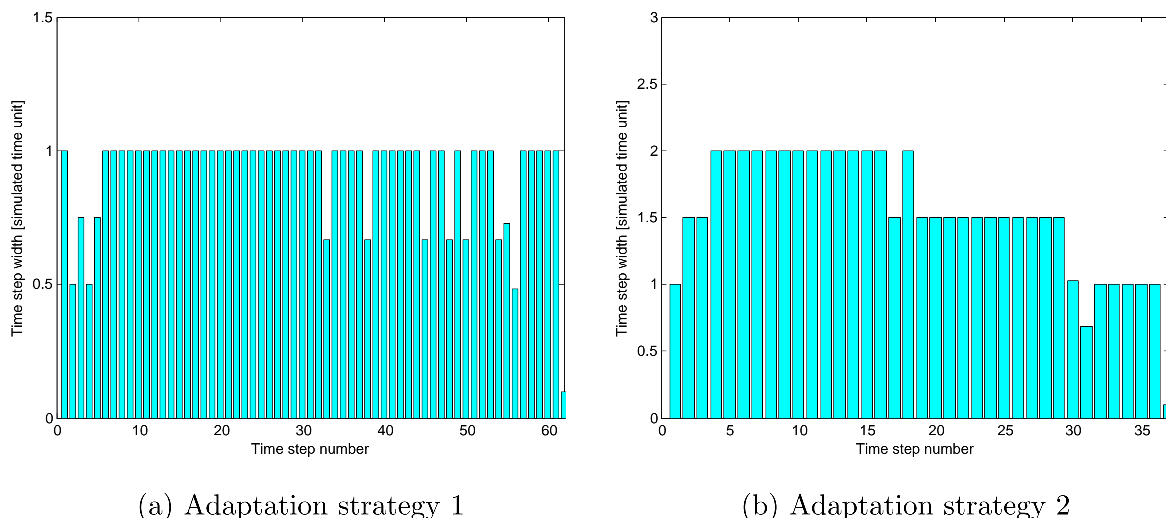
- *Strategy 1*: Large time step at the beginning; moderate changes of time step size when required; large range of steps allowed.
- *Strategy 2*: Very large time step at the beginning; moderate changes of time step size when required; small range of time steps allowed.

The experimental data contains records of the time step width and CPU time collected for each iteration of the time step loop for both strategies. Resulting time-series are shown in Figure 7.6.

Both experiments were carried out for the same input configuration. This means that the underlying simulation is identical and only the time steps are different depending on the applied strategy.

As one can see, the CPU time profiles for both adaptation strategies look similar and can be split qualitatively into the following phases: in the beginning low value, slowly increasing; followed by a fast increase leading to a high-value plateau and, finally, rapid decrease at the end.

The coarse grained temporal behavior described above is explained by differences in the convergence rate, i.e. number of solver iterations to reach forces equilibrium, which is due to changes in the underlying simulated metal-forming process. Additionally, the dynamics



(a) Adaptation strategy 1 (b) Adaptation strategy 2

Figure 7.7: Time step widths over the iterations of the time step loop

is also influenced by the time step adaptation strategies which result in more fine-granular spikes and variability in the magnitude as well as in the number of steps.

7.5.2 Automatic Analysis of Performance Dynamics with Periscope

As one could easily compute the total CPU time for strategy 1 is 32.69 min. For strategy 2 it is 36.7 min. Following a simple comparison, strategy 1 is a clear winner. In this study we will show how our performance dynamics analysis technique allows to derive additional insights and improve the results above.

In this study we evaluate performance dynamics of the first adaptation strategy. However, the question arises how to split the simulation process into analysis phases. A natural solution is to split simulation steps into phases with different convergence performance. Since we don't have the data explicitly providing such information, we use CPU time measurements to detect phases based on the variability in the metric values. Here the phases with slow convergence will result in peaks of the CPU time. Using Periscope Performance Dynamics Analysis Strategy and more specifically the Scale Space Filtering algorithm detecting such peaks is simple. For example, all we need in order to detect a peak is to search for "AB" sequences of qualitative descriptors at the maximum stability level obtained from the multi-scale qualitative representation. Therefore, we customize our analysis strategy to search only for the above qualitative pattern. In this example we skip evaluation of performance dynamics properties, since the available historical experimental data is not sufficient for the complete performance dynamics analysis.

Application of the automatic peaks detection in this particular case might be considered

Table 7.6: Convergence phases detected for adaptation strategy 1.

#	Iteration diapason	Sim.time diapason [sim. time units]	Cumulative CPU time [min.]	Simulation speed [sim. time units / min.]
1	[1,25]	(0,23.5]	4.72	5
2	[26,55]	(23.5,51.22]	25.77	1.08
3	[56,62]	(51.22,56.81]	2.19	2.56

as an overkill, since we have only two time-series which are both short and demonstrate quite simple temporal behavior. However, assuming the application to be run in parallel, where each process would show its own temporal dynamics due to domain decomposition, manual evaluation of series of CPU time on each process becomes very costly if possible at all.

Table 7.6 shows the detected phases of different convergence speeds for Strategy 1. In addition to the borders detected by Periscope and specified by the iterations of the time step loop, we further customize the search algorithm to provide the following additional information:

- Simulation time interval corresponding to the detected time step loop iteration phases. It is computed from the time step width records by taking cumulative sum of the widths from the beginning of the simulation until the time step corresponding to the beginning or the end of the converted interval.
- Cumulative sum of the CPU time within the phase
- Simulation speed. It is computed as a ratio of the simulation time inteval width over the cumulative CPU time during the phase.

As one could expect, the second phase corresponding to the high-value plateau in the second half of the CPU time measurements is the slowest according to the simulation speed parameter with only 1.08 simulation time units per minute of the CPU time. On the contrary, the first phase is the fastest with 5 units per minute. This proves another time our assumption that performance is not constant throughout the simulation.

But how does the second adaptation strategy performs during the simulation phases detected above? Table 7.7 shows the same parameters as in the previous one recalculated for the second strategy using the simulation time diapason as a reference for conversion. Since the simulation time domain is identical for both experiments, we can use simulation time diapason to convert it back to the iterations of the second strategy. Based on that we can compute the other parameters presented in the table.

Although, as expected, performance of the second strategy is worse in the second and the third phase, it surprisingly outperforms the first strategy in the first phase. This

Table 7.7: Convergence phases from adaptation strategy 1 converted to the time steps of the adaptation strategy 2.

#	Iteration diapason	Sim.time diapason [sim. time units]	Cumulative CPU time [min.]	Simulation speed [sim. time units / min.]
1	[1,13]	(0,24]	4	6
2	[14,30]	(24,51.02]	29.86	0.9
3	[31,37]	(51.02,56.81]	2.84	2.04

observation can be explained by the fact that Strategy 2 uses larger time steps in the beginning of the simulation. This appears to be more efficient for the state of the metal forming process simulated at this phase.

We use this knowledge to come up with a hybrid adaptation strategy which would follow strategy 2 in the beginning, namely until iteration 13, and then switch to strategy 1. By summing up the cumulative CPU times of the phases (strategy 2 in phase 1 and strategy 1 in phases 2 and 3) we get an estimated execution time for the hybrid strategy equal to 31.96 min, which is 2.23% faster in comparison to the strategy 1.

Chapter 8

Summary and Outlook

This work has presented a set of techniques which holistically tackle challenges of temporal performance dynamics in all the three major steps of the performance analysis process, namely, instrumentation, measurement and analysis. The main features of the proposed techniques are automation and time-dimension scalability.

Performance monitoring, as any other measurement process, requires insertion of probes in the monitored application. This is an important step of the performance analysis process called instrumentation. The granularity of the measurements, on one side, and the amount of introduced overhead, on the other side, are two mutually exclusive characteristics of the applied instrumentation. Here a trade-off has to be found, however, no common rule is available. This means that it has to be individually tuned, often manually, in each performance analysis study. Furthermore, for the majority of tools instrumentation configuration is applied for the whole measurement regardless of the temporal dynamics in the process.

In this thesis we presented a novel approach to automatically reaching such trade-offs. We propose a number of automatic instrumentation adaptation algorithms, called instrumentation strategies, which are able to tune insertion of probes according to a predefined objective. For the first time, the automatic instrumentation adaptation was proposed for the source-to-source direct instrumentation scheme. In order to quantify the amount of overhead introduced by the instrumentation, an overhead model was proposed for the probe functions with selective recording and partial canceling of overheads. Using the overhead estimations collected at runtime, the strategies evaluate overhead hypotheses against them. The hypotheses are formulated based on one of the objectives which guide the decision on the trade-off between the granularity and the overheads of measurements. Two objectives were proposed in this work: reduction of the total wall-clock time prolongation of the instrumented application (Total Overhead Reduction) and reduction of the overhead adsorbed in measurements (Prolog Overhead Reduction). The first objective is more strict and, in general, removes more instrumentation then the second one. It is more suited for the analysis scenarios when the prolongation (potentially un-even across

processes) may indirectly influence the measurements, for example in MPI wait state analysis. On the other hand, it might be too strict when only single-core performance is of interest. In such cases the second strategy is more appropriate. Finally, we propose the third strategy which allows dynamic adaptation of instrumentation guided by the runtime requirements of the on-line analysis process. This strategy allows to minimize overheads to the absolute necessary ones. Since our approach is based on the source-to-source instrumentation the adapted instrumentation requires recompilation of the affected source files. This has both advantages and disadvantages: instrumentation can be completely removed (in comparison to filtering out high-overhead regions at runtime), however, additional analysis time is wasted on recompilation and restarting the application.

In comparison to other techniques based on binary re-writing our approach features better granularity of measurements, simpler mapping of measurements to the source code and wider portability. The technique was implemented in Periscope and evaluated on a real-world application. The results show, that using the instrumentation adaptation strategies Periscope is able to detect performance inefficiencies which are lost due to high overheads otherwise.

With applications and hardware becoming more complex and, in particular, more dynamic, runtime performance variability adds an additional complexity dimension for application developers. Usage of tools is highly valuable and often un-avoidable for such scenarios. However, tools are faced with challenges as well when temporal performance is of interest. In particular, two aspects have to be addressed: how to collect and store temporal data in a time-dimension-scalable way, and how to extract and to convey the knowledge about dynamic performance degradations so that the analysis effort by the user is time-dimension-scalable as well.

It is obvious that, since the amount of memory available to the monitoring library is limited, temporal data cannot be collected for ever. Furthermore, the available memory on the compute node has to be shared between the application and the monitoring library. Our approach to tackling the issue called Online Dynamic Profile Analysis is twofold. First, we minimize the impact of collection and storing large temporal buffers in the place where it is the most critical, namely on the monitoring/application side. We achieve this by extending the dynamic profiling method, which is a light weight alternative to tracing for collecting coarse-grained temporal performance data. By adding on-line transmission of data samples to a remote analysis agent we decouple the size and overheads of storing data on the application side from the time dimension.

The bottleneck is, however, not completely removed, but shifted to another location. On the remote agent side we address the issue with on-line processing of raw temporal performance data in chunks of samples. This makes the amount of raw data buffered at the analysis agent a function of the chunk size and not the analysis time. Furthermore, by applying advanced automatic analysis techniques we extract relevant high-level knowledge about performance dynamics properties of the application and discard the raw measurements. The resulting properties are then merged along the time dimension as

well.

In order to reduce the user's efforts in comprehending dynamics captured in the raw temporal measurements we have proposed a first in its art automatic analysis technique. It is called Performance Dynamics Analysis Strategy and allows fully-automatic extraction of high-level performance dynamics properties from the raw temporal data. While designing the strategy, our goal was to bridge the semantic gap between the raw data represented by the tremendous amount of time-series and the "mental" model of performance dynamics employed by the user. We were able to achieve this goal by employing a set of algorithms from the fields of signal processing, computer vision and chemical process engineering orchestrated by an automatic rule-based inference engine. The algorithms allow both qualitative and quantitative analysis of time-series of performance data. We use wavelet analysis from the field of signal processing to quantify the amount of variability observed in the time-series. Scale-Space Filtering, based on incremental smoothing of the signal with Gaussian filter, is used to obtain a multi-scale representation in terms of signal's critical points. Such representation is native to the human perception, since these are the basis points we were taught to use when sketching a function. Furthermore, a unique property of the algorithm, adhering to similarities with the biological vision mechanism, is that it provides a quantitative measure for visual salience of the critical points. We use this together with a geometrical qualitative trend representation primitives borrowed from the chemical process engineering to perform qualitative summarization of the temporal dynamics observed in time-series of performance measurements. The representations, both qualitative and quantitative, allow formalization of typical patterns in terms of performance dynamics properties. These are then automatically evaluated by the Performance Dynamics Analysis Strategy.

The Online Dynamic Profile Analysis scheme and the Performance Dynamics Analysis Strategy were implemented in Periscope and evaluated with four real-world applications showing complex performance dynamics. We used manual analysis using value maps, since plotting every single time-series of measurements is impossible, as a reference. Already in the smallest case study with only 32 application processes the visual analysis of value maps hit the limits missing valuable performance dynamics patterns. Oppositely, the automatic analysis with Periscope was able to deliver insights obtained in the manual one plus the missed ones. Moreover, precise information about temporal location and severity of the degradations were reported. In the third case study with a long-running application the analysis was performed in chunks. Similarly, to the previous case, the automatic analysis was able to report degradations which were not visible during the manual investigation. In the final study, performance dynamics analysis was carried out for a commercial metal forming simulation software. We studied the dynamic performance of the application with two different simulation time step adaptation strategies. The first adaptation strategy was considered as a clear winner by the developers based on the wall-clock time. By investigating the dynamic behavior of the performance in both cases with Periscope we were able to see that the second strategy, although being slower for the whole execution, still outperforms the first one in the beginning of the simulation. By

combining two strategies we showed an improvement of 2.3% in comparison to the fastest one.

8.1 Future Work

The techniques presented in this thesis are one of the first steps towards automatic analysis of performance dynamics. While already being able to provide valuable insights, a number of possible areas for continuation and improvement exist. In particular, the grouping of performance dynamics properties can be further improved by performing multi-scale aggregation of patterns instead of concatenation and re-normalization of severity values. One way to achieve this is to keep the coarse-grained down-sampled approximations for each analysis chunk, which are then merged together in order to obtain a global picture of the temporal performance evolution.

The other interesting direction is to investigate dependencies between the detected degradations. Having multi-scale qualitative and quantitative representations proposed in this thesis, searching for dependencies promises to be more insightful then simply computing correlations between the metrics. In particular, adapting the Dynamic Time Warping (DTW) [48] algorithm to qualitative sequences used in this work would allow to estimate dependencies between the perceptually important features of the signal used to formalize performance dynamics degradations.

Bibliography

[1] 129.tera_tf SPEC MPI2007 benchmark description. http://www.spec.org/mpi2007/docs/129.tera_tf.html. Accessed: 2014-05-28.

[2] DEISA benchmarking and benchmark suite. http://www.deisa.eu/science/benchmarking/. Accessed: 2014-02-20.

[3] Dyninst. http://www.dyninst.org/. Accessed: 2014-04-28.

[4] Highly accurate finite element simulation for sheet metal forming. http://gns-mbh.com/indeed.html. Accessed: 2014-05-29.

[5] The human brain project. https://www.humanbrainproject.eu/. Accessed: 2014-01-28.

[6] Paradyn tools project. http://http://www.paradyn.org/. Accessed: 2014-02-20.

[7] Scalasca. http://www.scalasca.org/. Accessed: 2014-02-20.

[8] Score-p. http://www.vi-hps.org/projects/score-p/. Accessed: 2014-02-20.

[9] SuperMUC petascale system. http://www.lrz.de/services/compute/supermuc/. Accessed: 2014-05-28.

[10] Top 500 supercomputer sites. http://www.top500.org/list/2013/11/. Accessed: 2014-01-28.

[11] Vampir - performance optimization. http://www.vampir.eu/. Accessed: 2014-02-20.

[12] IEEE standard for software maintenance. *IEEE Std 1219-1998*, 1998.

[13] International standard - ISO/IEC 14764 IEEE Std 14764-2006 software engineering 2013; software life cycle processes 2013; maintenance. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 1–46, 2006.

[14] Dieter an Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, et al. Score-P: A unified performance measurement system for petascale applications. In *Competence in High Performance Computing 2010*, pages 85–97. Springer, 2012.

[15] Jean Babaud, Andrew P. Witkin, Michel Baudin, and Richard O. Duda. Uniqueness of the Gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence,*, (1):26–33, 1986.

[16] Bhavik R. Bakshi and George Stephanopoulos. Reasoning in time: Modeling, analysis, and pattern recognition of temporal process trends. *Advances in Chemical Engineering*, 22:485–548, 1995.

[17] B.R. Bakshi and G. Stephanopoulos. Representation of process trends - Part III. multiscale extraction of trends from process data. *Computers & Chemical Engineering*, 18(4):267–302, 1994.

[18] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, 2010.

[19] David Bohme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *39th International Conference on Parallel Processing (ICPP), 2010*, pages 90–100. IEEE, 2010.

[20] Holger Brunst, Dieter Kranzlmüller, Matthias S. Muller, and Wolfgang E. Nagel. Tools for scalable parallel program analysis: VampirNG, Marmot, and DeWiz. *International Journal of Computational Science and Engineering*, 4(3):149–161, 2009.

[21] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic phase detection and structure extraction of MPI applications. *International Journal of High Performance Computing Applications*, 24(3):335–360, 2010.

[22] J.T. Cheung and George Stephanopoulos. Representation of process trends - Part I. a formal representation framework. *Computers & Chemical Engineering*, 14(4):495–510, 1990.

[23] J.T. Cheung and George Stephanopoulos. Representation of process trends - Part II. the problem of scale and qualitative scaling. *Computers & chemical engineering*, 14(4):511–539, 1990.

[24] Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The dynamic probe class library-an infrastructure for developing instrumentation for performance tools. In *Proceedings 15th International Parallel and Distributed Processing Symposium.*, pages 7–pp. IEEE, 2001.

[25] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, pages 481–490, 2011.

[26] Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Träff. *Knowledge specification for automatic performance analysis: APART technical report*. Citeseer, 2001.

[27] Todd Gamblin. *Scalable performance measurement and analysis*. PhD thesis, Citeseer, 2009.

[28] Todd Gamblin, Bronis R. De Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Scalable load-balance measurement for SPMD codes. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–12. IEEE, 2008.

[29] Todd Gamblin, Bronis R. De Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Scalable load-balance measurement for SPMD codes. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–12. IEEE, 2008.

[30] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B.J.N. Wylie. Scalable collation and presentation of call-path profile data with CUBE. In *Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the International Conference ParCo 2007)*, pages 645–652. Citeseer, 2007.

[31] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian J.N. Wylie. Further improving the scalability of the Scalasca toolset. In *Applied Parallel and Scientific Computing*, pages 463–473. Springer, 2012.

[32] Michael Gerndt. Specification of performance properties of hybrid programs on hitachi SR8000. Technical report, Peridot Technical Report, TU München, 2002.

[33] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced techniques for performance analysis. In *PARCO*, pages 15–26. Citeseer, 2005.

[34] Michael Gerndt and Michael Ott. Automatic performance analysis with Periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[35] Paul Gibbon. *PEPC: Pretty efficient parallel Coulomb-solver*. FZJ-ZAM, 2003.

[36] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[37] Andreas Knupfer and Wolfgang E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing, 2005. ICPP 2005.*, pages 165–172. IEEE, 2005.

[38] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[39] Jan J. Koenderink. The structure of images. *Biological cybernetics*, 50(5):363–370, 1984.

[40] Tony Lindeberg. Scale-space for discrete signals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(3):234–254, 1990.

[41] Tony Lindeberg. Generalized Gaussian scale-space axiomatics comprising linear scale-space, affine scale-space and spatio-temporal scale-space. *Journal of Mathematical Imaging and Vision*, 40(1):36–81, 2011.

[42] Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi. Extending Scalascas analysis features. In *Tools for High Performance Computing 2012*, pages 115–126. Springer, 2013.

[43] S. Mallat. *A wavelet tour of signal processing*. Academic press, 1999.

[44] Matija Mihelcic, Helmut Wenzl, and Kurt Wingerath. *Flow in czochralski crystal growth melts*. Forschungszentrum, Zentralbibliothek, 1992.

[45] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[46] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. *Towards a performance tool interface for OpenMP: An approach based on directive rewriting*. Forschungszentrum, Zentralinst. für Angewandte Mathematik, 2001.

[47] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *PARCO*, pages 637–644. Citeseer, 2007.

[48] Meinard Müller. Dynamic time warping. *Information retrieval for music and motion*, pages 69–84, 2007.

[49] M.S. Müller, M. van Waveren, R. Liebermann, B. Whitney, H. Saito, K. Kalyan, J. Baron, B. Brantley, Ch. Parrott, T. Elken, et al. SPEC MPI2007 – an application benchmark for clusters and HPC systems. *ISC2007*, 2007.

[50] Ole Møller Nielsen and Markus Hegland. Parallel performance of fast wavelet transforms. *International Journal of High Speed Computing*, 11(01):55–74, 2000.

[51] D.B. Percival and A.T. Walden. *Wavelet methods for time series analysis*, volume 4. Cambridge University Press, 2006.

[52] Ventsislav Petkov. *Automatic Performance Engineering Workflows for High Performance Computing*. PhD thesis, TUM, 2014.

[53] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21. ACM, 2003.

[54] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2):105–121, 2008.

[55] Jerome M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.

[56] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[57] Z. Szebenyi, B. Wylie, and F. Wolf. Scalasca parallel performance analyses of PEPC. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 305–314. Springer, 2009.

[58] Zoltán Szebenyi, Felix Wolf, and Brian J.N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 37. ACM, 2009.

[59] Zoltán Péter Szebenyi. *Capturing Parallel Performance Dynamics*, volume 12. Forschungszentrum Jülich, 2012.

[60] Saul A. Teukolski, Brian P. Flannery, William H. Press, and William T. Vetterling. *Numerical Recipes in FORTRAN - The Art of Scientific Computing*. University Press, 1989.

[61] Andrew P. Witkin. Scale-space filtering: A new approach to multi-scale description. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'84.*, volume 9, pages 150–153. IEEE, 1984.