TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT INFORMATIK

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

Concepts for In-memory Event Tracing

Runtime Event Reduction with Hierarchical Memory Buffers

Michael Wagner
geboren am 18. Mai 1984 in Bad Schlema

Gutachter: Prof. Dr. Wolfgang E. Nagel
Technische Universität Dresden

Prof. Dr. Felix Wolf
Technische Universität Darmstadt

Dresden, 1. März 2015

Abstract

This thesis contributes to the field of performance analysis in High Performance Computing with new concepts for in-memory event tracing.

Event tracing records runtime events of an application and stores each with a precise time stamp and further relevant metrics. The high resolution and detailed information allows an in-depth analysis of the dynamic program behavior, interactions in parallel applications, and potential performance issues. For long-running and large-scale parallel applications, event-based tracing faces three challenges, yet unsolved: the number of resulting trace files limits scalability, the huge amounts of collected data overwhelm file systems and analysis capabilities, and the measurement bias, in particular, due to intermediate memory buffer flushes prevents a correct analysis.

This thesis proposes concepts for an in-memory event tracing workflow. These concepts include new enhanced encoding techniques to increase memory efficiency and novel strategies for runtime event reduction to dynamically adapt trace size during runtime. An in-memory event tracing workflow based on these concepts meets all three challenges: First, it not only overcomes the scalability limitations due to the number of resulting trace files but eliminates the overhead of file system interaction altogether. Second, the enhanced encoding techniques and event reduction lead to remarkable smaller trace sizes. Finally, an in-memory event tracing workflow completely avoids intermediate memory buffer flushes, which minimizes measurement bias and allows a meaningful performance analysis.

The concepts further include the *Hierarchical Memory Buffer* data structure, which incorporates a multi-dimensional, hierarchical ordering of events by common metrics, such as time stamp, calling context, event class, and function call duration. This hierarchical ordering allows a low-overhead event encoding, event reduction and event filtering, as well as new hierarchy-aided analysis requests.

An experimental evaluation based on real-life applications and a detailed case study underline the capabilities of the concepts presented in this thesis. The new enhanced encoding techniques reduce memory allocation during runtime by a factor of 3.3 to 7.2, while at the same do not introduce any additional overhead. Furthermore, the combined concepts including the enhanced encoding techniques, event reduction, and a new filter based on function duration within the Hierarchical Memory Buffer remarkably reduce the resulting trace size up to three orders of magnitude and keep an entire measurement within a single fixed-size memory buffer, while still providing a coarse but meaningful analysis of the application.

This thesis includes a discussion of the state-of-the-art and related work, a detailed presentation of the enhanced encoding techniques, the event reduction strategies, the Hierarchical Memory Buffer data structure, and a extensive experimental evaluation of all concepts.

Contents

1 Introduction **1**

2 State-of-the-art in Event-based Performance Analysis **5**

2.1 Performance Analysis . . . 5
2.2 Performance Analysis Tools Overview . . . 8
2.3 Event-based Trace Analysis Tools . . . 10
2.3.1 The Vampir Toolset . . . 10
2.3.2 The Paraver Toolset . . . 16
2.3.3 The Scalasca Toolset . . . 18
2.3.4 The Tuning and Analysis Utilities (TAU) . . . 20
2.3.5 Score-P and the Open Trace Format 2 . . . 21
2.4 Challenges in Event-based Tracing and Related Work . . . 24
2.4.1 Scalability . . . 24
2.4.2 Data Volumes . . . 25
2.4.3 Measurement Bias . . . 28
2.5 Open Challenges and In-memory Event Tracing . . . 30
2.6 Summary . . . 30

3 Concepts for In-memory Event Tracing **33**

3.1 In-memory Event Tracing . . . 33
3.2 Selection and Filtering . . . 35
3.3 Enhanced Encoding Techniques . . . 36
3.3.1 Binary Event Representation . . . 37
3.3.2 Splitting of Timing Information and Event Data . . . 38
3.3.3 Leading Zero Elimination . . . 39
3.3.4 Delta Encoding . . . 39
3.3.5 Event Distribution and Encoding Implications . . . 40
3.3.6 Timer Resolution Reduction . . . 43
3.4 Event Reduction . . . 44
3.4.1 Reduction by Order of Occurrence . . . 45
3.4.2 Reduction by Event Class . . . 46
3.4.3 Reduction by Calling Depth . . . 47
3.4.4 Reduction by Duration . . . 50
3.4.5 Requirements for Event Reduction . . . 52
3.5 Summary . . . 53

4 The Hierarchical Memory Buffer **55**
4.1 Memory Event Representation . . . 55
4.1.1 Flat Continuous Event Representation . . . 55
4.1.2 Flat Partitioned Event Representation . . . 56
4.1.3 Hierarchical Event Representation . . . 58
4.2 The Hierarchical Memory Buffer Data Structure . . . 60
4.3 Construction of the Hierarchical Memory Buffer . . . 61
4.4 Reduction Techniques with the Hierarchical Memory Buffer . . . 65
4.4.1 Reduction by Order of Occurrence . . . 65
4.4.2 Reduction by Event Class . . . 66
4.4.3 Reduction by Calling Depth . . . 68
4.4.4 Reduction by Duration . . . 69
4.5 Analysis Techniques for the Hierarchical Memory Buffer . . . 70
4.5.1 Linear Time Iterator . . . 70
4.5.2 Forward Traversal . . . 73
4.5.3 Statistical Summaries . . . 74
4.5.4 Timeline Visualisation . . . 75
4.5.5 Message Matching . . . 75
4.6 Message Matching on Incomplete Communication Data . . . 76
4.6.1 Message Matching Approaches . . . 76
4.6.2 Identification of Missing Communication Events . . . 78
4.6.3 Adapted Message Matching . . . 79
4.7 Adaption to Sampling . . . 80
4.8 Summary . . . 83

5 Evaluation and Case Study **85**
5.1 Methodology and Target Applications . . . 85
5.2 Enhanced Encoding Techniques . . . 87
5.2.1 Runtime Memory Allocation . . . 87
5.2.2 Runtime Overhead . . . 92
5.3 The Hierarchical Memory Buffer . . . 94
5.3.1 Determine an Ideal Memory Bin Size . . . 94
5.3.2 Reduction of Hierarchy Partitions . . . 98
5.3.3 Reduction by Duration . . . 99
5.3.4 Analysis Techniques . . . 103
5.3.5 Message Matching on Incomplete Communication Data . . . 104
5.4 Case Study: The Molecular Dynamics Package Gromacs . . . 106
5.4.1 The Molecular Dynamics Package Gromacs . . . 106
5.4.2 The Bias Caused by Intermediate Buffer Flushes . . . 107
5.4.3 In-memory Event Tracing for Long Application Runs . . . 109
5.5 Summary . . . 114

6 Conclusion and Outlook **115**

Bibliography **119**

List of Figures **129**

List of Tables **133**

1 Introduction

It is beneath the dignity of excellent men to waste their time in calculation when any peasant could do the work just as accurately with the aid of a machine.

GOTTFRIED WILHELM LEIBNIZ

When Gottfried Wilhelm Leibniz presented the *Stepped Reckoner*, the first calculation machine, to the Royal Society of London in 1673 he paved the way to today's computers. While his and all following calculation machines of the next two centuries did not use electronics but were the product of precise engineering, he already constituted his machine *supra hominem* – superior to humans – since it outperformed humans in speed, as well as accuracy, for large calculations. With the advent of electronic computers, first based on electromechanical relays and later on vacuum tubes and transistors, the capabilities and performance of machine-aided computing began to grow rapidly. Nowadays, computing devices are omnipresent and an integral part of many aspects of life. Particularly in science and research, computer-aided simulation has become indispensable and is considered the third cornerstone of scientific methodology besides theory and experiment.

Beyond everyday computing devices, High Performance Computing (HPC) systems provide enormous computational resources to support large-scale simulations in leading-edge scientific research such as the Human Brain project [MML+11], climate and weather prediction, or DNA and cancer research. Today, High Performance Computing typically includes a large number of processing elements working jointly on a computationally intensive problem. For the next milestone, exa-scale supercomputers capable of $\mathcal{O}(10^{18})$ floating point operations per second, this approach is very likely to persist. For the foreseeable future Moore's law [Moo65] is expected to endure, however, limitations in clock frequency, instruction level parallelism, and energy density drive the further increase in the number of processing elements. In addition, supercomputing hardware is strongly influenced by economy-driven developments in the off-the-shelf market, as shown, for instance, by the integration of accelerators from graphic cards. The history of TOP500 [Top14] systems shows that not all system characteristics improve at the same speed as computing power. Critical properties are main memory bandwidth and latency, the amount of memory per core, I/O capabilities, as well as energy consumption [BBC+08]. Consequently, supercomputers targeting the exa-scale barrier are likely to be specialized solutions with tremendous computational power but also many constraints that have a considerable impact on efficient software development.

Parallel software that scales to the exa-scale level implies the identification, distribution, and synchronization of millions of subproblems that can be computed autonomously. Any computationally intensive problem requires the decomposition in subtasks, whose partial results must be accumulated efficiently to the overall solution. Writing software for systems of this scale is demanding and involves hybrid and new programming models, accelerated computing, and energy considerations. Hence, appropriate supporting tools, such as debuggers and performance analyzers, are inevitable to develop applications that utilize the enormous capabilities of current and future HPC systems.

Performance analysis tools assist developers not only in identifying performance issues in their applications but also in understanding their behavior on complex and heterogeneous systems. Such tools gather information about the behavior of an application during runtime by either recording runtime events or by periodically sampling its current state. While sampling approaches rely on their sampling frequency to gain information about an application, event-based monitoring tools record information if specific predefined events occur, for instance, entering and leaving a function.

Information gathered from samples or events can be aggregated to summarized information about different performance metrics (profiling) or stored individually by keeping the precise time stamp and further specific metrics for each event (tracing). Profiling with its nature of summarization decreases the amount of data that needs to be stored during runtime. However, profiles may lack critical information and hide dynamically occurring effects. In contrast, event tracing records each event of a parallel application in detail and allows an exact reconstruction of the application behavior. Thus, it enables capturing the dynamic interaction between thousands of concurrent processing elements and the identification of outliers from the regular behavior. Such detail comes with the cost that event-based tracing frequently results in huge data volumes even though single events are rather small. In fact, the large amount of collected data, in particular, for massively parallel or long running applications is one of the most urgent challenges for event-based monitoring tools.

In both dimensions event tracing is already pushing against the limits of today's and, most likely, also tomorrow's systems. Since the collected data is usually stored in one file per processing element, the number of resulting trace files is increasing with the number of recorded processing elements. While HPC parallel file systems are highly optimized for data throughput, the simultaneous creation of hundreds of thousands or even millions of event tracing files overwhelms any parallel file system and the aggregated size of the resulting trace files quickly swallows up storage capacities. Next to that, the recorded event data is typically buffered before it is written to the file system to reduce expensive file system interactions. Whenever such an internal memory buffer is exhausted, the content is transferred to the file system; usually in an unsynchronized fashion. Such uncoordinated intermediate memory buffer flushes during a measurement introduce extensive bias and lead to a falsification of the recorded program behavior. Much like system noise this effect is increasing with higher scales.

Another way to circumvent these constraints is an *in-memory event tracing workflow* that completely omits file system interaction. Keeping recorded event data in main memory for the complete workflow would not only bypass the limitations in the number of file handles, moreover, it would eliminate the overhead of file creation, writing and reading altogether. In addition, an in-memory workflow would exclude the bias caused by non-synchronous intermediate memory buffer flushes during a measurement run. Furthermore, such a workflow allows entirely new features in event tracing, such as an event-based online performance analysis workflow.

But there is one catch. Keeping event data in main memory for a complete measurement requires that recorded data fits into a single memory buffer of an event tracing library. Unfortunately, measurement runs may collect hundreds of megabytes up to gigabytes of data per processing element. To make things worse, the part of the main memory left to store the data is rather small, since most applications utilize main memory intensively. This thesis is dedicated to the challenge of fitting an entire measurement of arbitrary length and scale into a single fixed-sized memory buffer for each processing element and, therefore, setting the premise for an in-memory event tracing workflow.

Contribution of this Thesis

The contributions of this thesis are novel concepts to enable an in-memory event tracing workflow. These concepts are divided in two central parts:

Enhanced encoding techniques and *strategies for event reduction* that dynamically adapt trace size during runtime to the given memory allocation form the first central part of the contributions of this thesis. The combination of both allows keeping the data of an entire measurement within a single fixed-sized memory buffer and, therefore, enable an in-memory event tracing workflow. First, such an in-memory tracing workflow not only bypasses the limitations of current parallel file systems but eliminates the overhead of file system interaction altogether. Second, the enhanced encoding techniques and event reduction result in remarkably smaller trace sizes. Furthermore, the in-memory workflow completely avoids intermediate memory buffer flushes and, therefore, minimizes measurement bias, which allows a feasible tracing of long running applications.

The *Hierarchical Memory Buffer* is the second central contribution of this thesis. The Hierarchical Memory Buffer is a new data structure that uses hierarchy information, such as calling depth or event class, to presort events according to these hierarchy attributes. It allows performing the aforementioned event reduction operations with minimal overhead. Furthermore, such a hierarchy-based event representation allows new event filter operations, unfeasible with a traditional flat, continuous memory buffer layout. Such a new filter method is a filtering based on the duration of code regions, which eliminates all short-running functions while keeping outliers important for performance analysis. In addition, several typical analysis requests can benefit from a hierarchy-aided traversal of recorded event data.

Organisation of this thesis

The next chapter, *State-of-the-art in Event-based Performance Analysis*, provides an overview of the state-of-the-art in event-based performance analysis and performance analysis tools. Furthermore, tools for event-based trace recording and analysis are discussed in more detail. On the basis of three current challenges the chapter introduces related work and connects the contributions of this thesis.

The chapter on *Concepts for In-memory Event Tracing* specifies prerequisites for an in-memory event tracing workflow and defines three key steps to keep an entire measurement within a single fixed-size memory buffer. These three key steps are selection and filtering, enhanced encoding, and event reduction. New methods for each step are presented.

The chapter *The Hierarchical Memory Buffer* introduces the Hierarchical Memory Buffer, a data structure that allows to perform event reduction operations with minimal overhead. It examines algorithms for the construction of this data structure, as well as the application of the event reduction strategies and typical analysis techniques. Furthermore, the computational complexity of all algorithms is discussed.

The chapter *Evaluation and Case Study* presents an evaluation of the enhanced encoding techniques and the Hierarchical Memory Buffer data structure including its capabilities to support the event reduction strategies. In addition, a detailed case study demonstrates the benefits of the combined approach for a real-life application.

The final chapter completes this thesis with a conclusion and an outlook to future research.

2 State-of-the-art in Event-based Performance Analysis

This chapter provides an overview of the state-of-the-art in event-based performance analysis and performance analysis tools. Furthermore, tools for event-based trace recording and analysis are discussed in more detail. On the basis of three current challenges this chapter introduces related research and connects the contributions of this thesis.

2.1 Performance Analysis

As High Performance Computing (HPC) systems are getting more and more powerful, they are getting more and more complex, as well. Besides the already intricate processing core designs that require a consideration of hierarchical memory accesses via multi-level caches, pipelined instruction execution, branch prediction and execution, and built-in vector units; parallel systems that use thousands or even millions of these compute cores demand additional consideration of parallel execution, network, system topology, and hardware accelerators – to name only a few. On top of the complex hardware is a complementary complex software stack that includes batch systems and application scheduling, resource distribution, and a variety of parallel paradigms such as message passing, threading, partial global address space, and paradigms to use hardware accelerators, which, more and more often, promise to be efficient only when combined correctly. Developing applications that utilize the enormous capabilities of these complex systems requires a continuous process of optimization – even for well-established software projects that have been in development for decades.

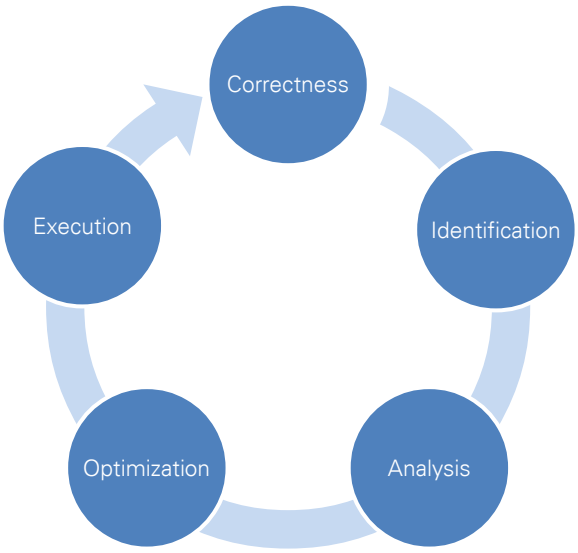


Figure 2.1: Optimization cycle: starting with correctness checking, identifying inefficient program phases, analyzing these program phases, optimizing the application, and execution.

The optimization process for parallel applications consists of five basic steps that are shown in Figure 2.1. The first step is debugging and correctness checking to ensure a correct program execution. The second step is to get a coarse view on the application behavior and identify program phases that contain irregular or inefficient behavior. These program phases can then be reviewed and analyzed in detail. The gained information can be used to optimize either the algorithm itself or its adaption to the current hardware and software stack and, finally, execute the revised application.

Within this optimization cycle, performance analysis covers the identification and analysis steps and provides input for optimization. Appropriate performance analysis tools assist developers not only in identifying performance inefficiencies but also in understanding the behavior of their applications on the complex and heterogeneous HPC systems. Furthermore, they help to analyze the performance inefficiencies and provide insight into the exact course of events during the application runtime.

The variety of concepts and according tools can be categorized by their approach for capturing, recording, and presentation of performance information [Jai91], as shown in Figure 2.2. Within the first stage, data capturing, there are two different methods. *Sampling* approaches interrupt a running application at arbitrary points (usually fixed intervals) and record the current state of execution. Whereas *event-based* methods record the state changes in the program execution, so called events, e.g., entering or leaving a code region. While the accuracy of sampling approaches relies on their frequency, event-based methods record every predefined runtime event, which means, the accuracy can be defined a priori. Vice-versa, sampling approaches can regulate the introduced measurement overhead and the memory allocation with their sampling frequency, while the overhead and memory allocation of event-based methods correlates with the frequency of the predefined runtime events.

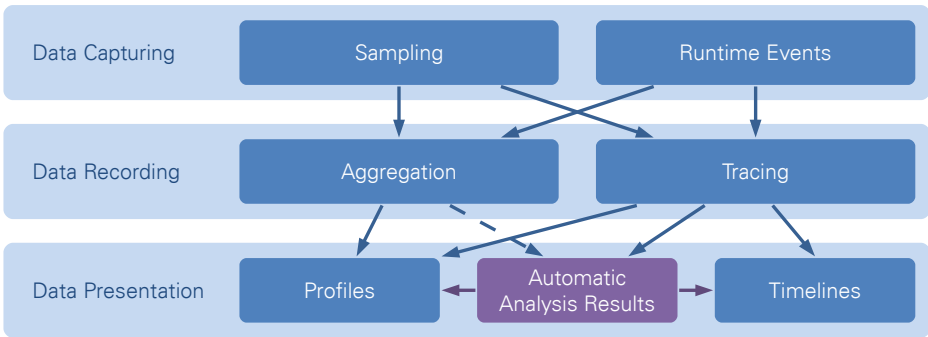


Figure 2.2: The three stages of performance analysis: data capturing, data recording, and data presentation. Arrows indicate possible transitions between the stages.

The generated data from both approaches can then be either *aggregated* to summaries about different performance metrics or stored individually by keeping the precise time stamp and further specific metrics for each sample or event, called *tracing*. Aggregation approaches (also called profiling approaches[1]) immediately combine the data of new samples or events with previously recorded data. With its nature of summarization this method decreases the amount of data that needs to be stored during runtime. In contrast, tracing records each sample or event individually. It keeps every recorded state (sample) or state change (event) intact, which allows an exact reconstruction of the program behavior – with an accuracy

[1]Since aggregated data from samples or events can only be presented in the form of profiles, the method of aggregating data to summaries is often referred to as profiling, as well.

based on either the sampling frequency or the predefined events. Consequently, tracing data includes aggregated data because aggregated summaries always can be computed from a trace.

The aggregated and traced data can be presented in three different ways: profiles, timelines, and automatically generated analysis results. Profiles display a summarization of one or more performance metrics over the entire program runtime or separately for defined program phases. These summaries can be represented as plain text, tables or charts. A typical example is the distribution of the overall program runtime over the different code regions or the total number of invocations for each code region. Profiles can be derived directly from aggregated data or computed from traces. Timelines display the initially recorded program states or state changes along a time axis and, therefore, show the exact state of the application at any give time. Since this approach requires the exact states or state changes it can only be derived from tracing data. Next to profiles and timelines that represent the recorded data, there are approaches that evaluate the recorded data and enrich the presentation of either profiles or timelines with results of an automatic analysis. Automatic analyses usually focus on the detection of typical inefficient patterns or the evaluation of specific performance metrics that might hint inefficient behavior.

While there are multiple possible combinations within these three stages, there are two well-established paths: *profiling*, which usually refers to aggregated events presented as profiles, and *event tracing*, which visualizes event traces in the form of timelines or leads to an automatic analysis. Both approaches differ greatly in memory allocation and extractable information. Due to the summarization, the memory allocation for profiling correlates only with the number of different performance metrics that are recorded, whereas, the memory allocation of event tracing correlates with the number of runtime events. While single events are rather small, event tracing frequently results in huge data volumes. In fact, the large amount of collected data, in particular, for massively parallel or long running applications is one of the most urgent challenges for event-based monitoring tools.

Despite this drawback, event tracing is an essential technique since the exact communication behavior and many performance inefficiencies can only be identified with event tracing. Two prominent performance issues are excessive wait time in communication operations and load imbalances [Boe14]. Excessive wait time in communication, so called wait states, are usually caused by two communication partners that enter a communication operation not properly synchronized, i.e., either the sender or receiver enters to late. While profiling can only hint inefficient communication behavior in general based on the total time spent in communication operations, event tracing can uncover each individual inefficient communication operation and its cause [Boe14]. The conglomeration of these wait states reveals the critical path in the execution of an application, as well. Load imbalances can occur in different forms. Figure 2.3 shows an example that helps to distinguish between the analysis capabilities of event tracing and profiling for load imbalances. The left side depicts a timeline visualization of a load imbalance with four synchronizations depicted by the vertical grey lines. On the right side, a profile visualization shows the aggregated runtime for each process. The upper part represents a static load imbalance, i.e., the process that takes longer for the execution (red) is the same in each program phase. In contrast, the bottom part represents a dynamic load imbalance where the processes that causes the delay is different in each program phase. The timeline based on event tracing reveals both imbalance patterns, while the profile based on aggregated data only reveals the static imbalance, i.e., in this case it is impossible to detect the dynamic load imbalance. In addition, event tracing enables the detection of the critical path in both cases, which is in this example the sequence of the red program phases in the timeline.

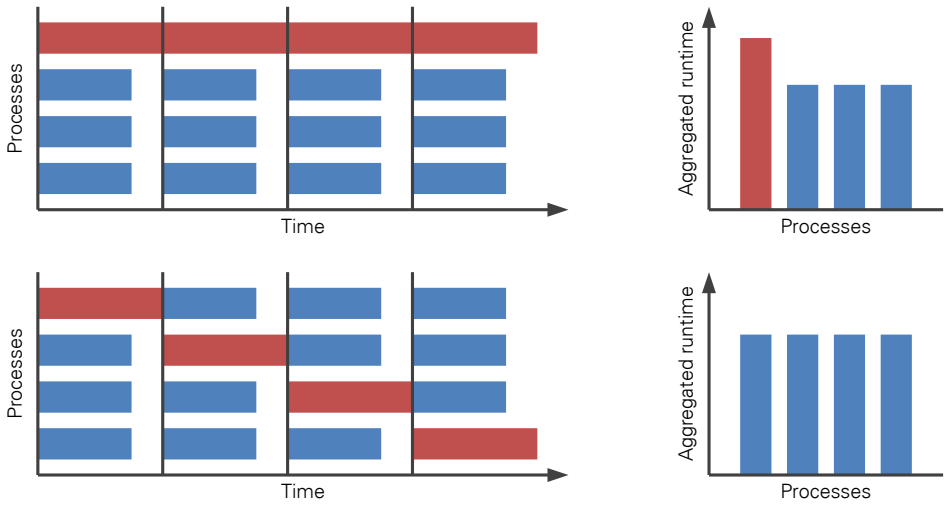


Figure 2.3: Analysis capabilities of tracing vs. profiling: static (top) and dynamic (bottom) load imbalance presented as timelines (left) and profiles (right).

Research in performance analysis also hints towards a trend that the gap between both approaches is shrinking. Profiling approaches, on the one hand, use techniques such as phase-based profiling [MSM05, CBL07] or call-path/call-graph profiling [GKM82, SWW09] to gain more information than with flat profiles over the entire runtime. On the other hand, event tracing approaches try to either reduce the number of predefined runtime events, e.g., only record the communication behavior, or try to reduce the number of stored events to cope with the tremendous data volumes.

2.2 Performance Analysis Tools Overview

After an introduction to the field of performance analysis, this section provides on overview about established performance analysis tools and their approaches. Figure 2.4 shows a classification of these performance analysis tools. Since, data aggregation usually leads to a profile presentation and tracing data is usually visualized with timelines, the figure uses the terms profiling and tracing for an easier-to-read presentation. While there are many more academic and commercial tools that focus on single aspects of performance (e.g., only communication [VM01, NRM+09]), this classification includes tools that use a general approach and analyze multiple performance relevant metrics.

Event-based Profiling Tools

Profiling tools based on events, also called instrumentation profiling tools, are gprof [GKM82], IPM [FWS10], and Periscope [BPG10].

In this list, *gprof* is one of the oldest and probably most used tools, since it is included in many Unix systems. It supports event-based call-graph profiling but also profiling based on sampling. Gprof results in a basic text output that shows typical profile information such as time spent in code regions or their number of invocations.

The *Integrated Performance Monitoring (IPM)* is a collaborative project between the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory and the San

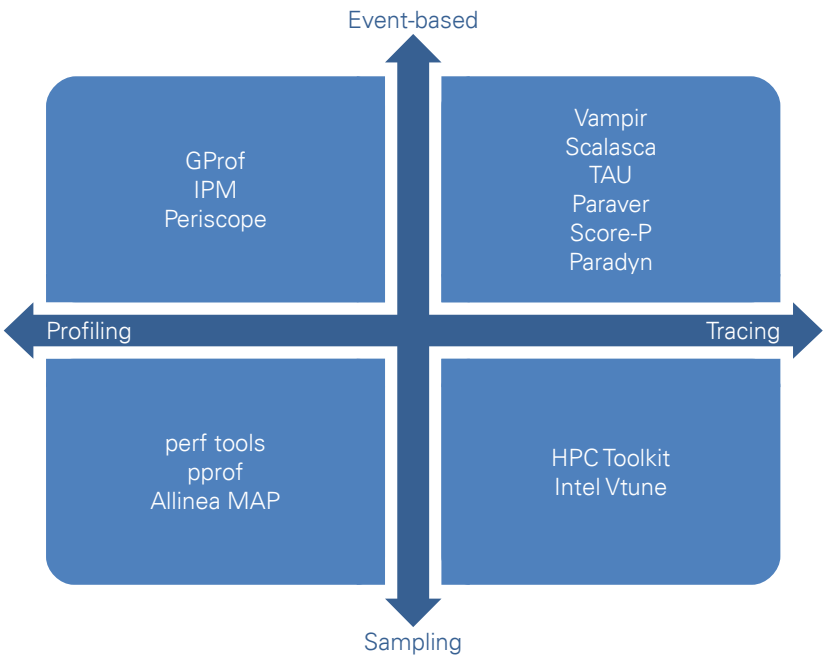


Figure 2.4: Classification of performance analysis tools.

Diego Supercomputer Center (SDSC). It focusses primarily on communication, computation, and I/O and provides an output in the form of tables, charts and plots.

Periscope is an automatic performance analysis tool developed at Technische Universität München, Germany. It consists of a front-end integrated in the Eclipse IDE[2] and a hierarchy of communication and analysis agents. Each of the analysis agents searches autonomously for defined patterns that indicate inefficient behavior in a subset of the application processes. The results are fed back to the Eclipse IDE and relate to the according source code parts.

Statistical Profiling Tools

Profiling tools that aggregate sampling data, also called statistical profiliers, are the Linux perf tools [Wea14], pprof [GG14], Allinea Map [All14] and the aforementioned gprof [GKM82].

The *Linux perf tools* (previosly Performance Counters for Linux (PCL)) is part of the linux kernel since version 2.6.31, which allows statistical profiling of the entire system, both, kernel and user space code.

The *Google performance tools* include a profiling tool based on sampling. With the also contained pprof tool the gathered data can be translated to a plain text output or a graphic call graph annotated with timing information [Ghe08].

Allinea's MAP tool presents itself as the most elaborated of the statistical profilers. It implements adaptive sampling rates and provides a sophisticated graphical user interface. However, the tool is proprietary and its approach is not scientifically evaluated.

[2]Eclipse Integrated Development Environment, www.eclipse.org

Sampling-based Tracing Tools

Tracing tools based on samples are the HPCToolkit [ABF+10] and Intel's VTune [Rei05]. Furthermore, there are event-based tracing tools that support sampling, too, e.g., Paraver and Score-P (see below).
The *HPCToolkit*, developed at Rice University, is a sampling-based tracing tool that supports three different visualization methods: code-centric (i.e., a call-path profile), thread-centric, and time-centric (i.e., a timeline) [MCA+14]. A unique feature of HPCToolkit is its ability to combine the recovered static program structure with dynamic calling context information to attribute performance metrics to calling contexts, procedures, loops, and inlined instances of procedures [TMCF09].
Intel's VTune in its current version as Intel VTune Amplifier 2015 [Int14a] is a proprietary tool that provides basic profiles and timelines.

Event-based Tracing Tools

The fourth category, event-based tracing tools, contains analysis tools such as Vampir [NAW+96], the Scalasca toolset [GWW+10], the Tuning and Analysis Utilities (TAU) [SM06], the Paraver toolset [SLGL10], and the Paradyn toolset [MCC+95]. Furthermore, there is the measurement infrastructure Score-P [KRM+12], which serves as a unified monitoring system for the analysis tools Vampir, Scalasca, and Tau. Because they are the primary target of the contributions presented in this thesis, these tools are covered in more detail in the next section.

2.3 Event-based Trace Analysis Tools

This section narrows down the focus on well-established event-based performance analysis tools and their general approaches. While there are many more academic and proprietary approaches [MCLD01], for instance, Paradyn [MCC+95], the Intel Trace Analyzer [Int14b], OpenSpeedShop [SGM+08], and Jumpshot [ZLGS99], this section focuses on tools that represent a distinguished class of features. Because other tools share similar approaches, they are not covered separately. Furthermore, this section provides a basis for the discussion of the current challenges in event-based tracing presented in the next section and allows to assess requirements for enhancements to the current approaches.

2.3.1 The Vampir Toolset

Vampir (Visualization and Analysis of MPI Resources) [KBD+08] is a well-proven and widely used toolset for event-based performance analysis in the high performance computing community. It consists of the Vampir trace visualizer [NAW+96, BN03, BWNH01] and the VampirTrace measurement environment [MKJ+07, ZIH14]. The Vampir visualizer is available as a commercial product since 1996. Its development started at the Center for Applied Mathematics of the Research Center Jülich, Germany and is continued at the Center for Information Services and High Performance Computing (ZIH) of the Technische Universität Dresden, Germany. Its counterpart, the VampirTrace measurement environment is available as open source software since 2006. It supports all major parallel paradigms and accelerator APIs simultaneously, e.g., message passing (MPI), threading (OpenMP, Pthreads), and accelerator APIs (CUDA, OpenCL). Within the collaborative project SILC [SILC09] the unified measurement infrastructure Score-P was developed, which is replacing VampirTrace now (see Section 2.3.5).

Today, the Vampir trace visualizer includes a scalable, distributed analysis architecture called VampirServer [BNM03, BMSB03]. The VampirServer architecture enables the scalable processing of both, large amounts of trace data and large numbers of processing elements. This architecture consists of a visualization client, a master process, and a number of distributed workers (see Figure 2.5). The client is intended to run on a user's local system and visualizes the display information received from the server. The master process of the server handles the requests from the client and distributes partial requests to the workers. The workers evaluate disjoint parts of the trace data, usually a subset of locations of the monitored application, and send the results to the master. The master communicates these results in the form of already composed display information to the client, hence, it requires only a moderate network connection between client and server. Small, local traces can also be evaluate directly by the client.

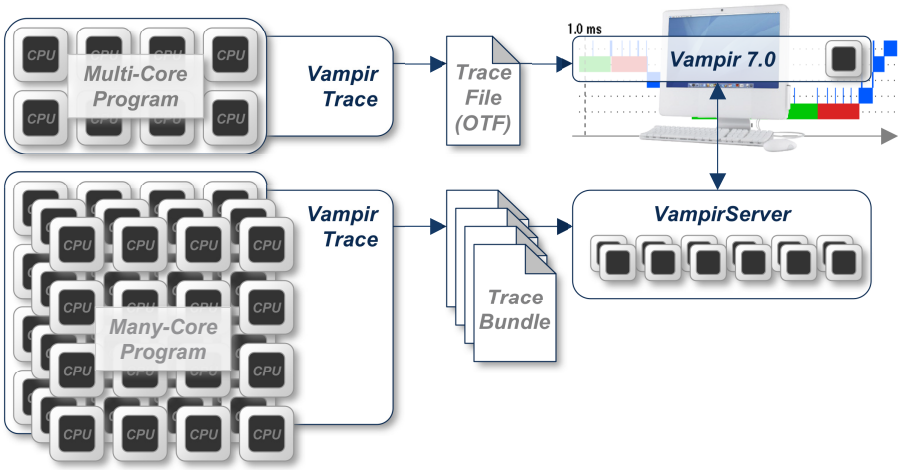


Figure 2.5: Vampir/VampirTrace architecture overview. VampirTrace (left) records events from the parallel application. The resulting trace file is processed either directly by the Vampir client or by the VampirServer in the case of large parallel programs (taken from [BHJH10]).

As stated before, event-based tracing tools present the tracing data in the form of timelines along with summarized profile information and automatically derived analysis results. The main visualization approaches of Vampir, exemplary for many similar tools, are detailed below.

Master Timeline Display

A master timeline visualization generates a visual representation of the application behavior over time in the form of a space-time diagram. It represents the active code region over time for each location on the horizontal axis and the selected locations on the vertical axis. Whereas each code region or group of code regions is marked as a segment of a location bar with a specific color for the time it is active. Communication and interaction between locations are represented by arrows and lines. For each element of the master timeline detailed context information is available, when selected.

While the initial representation of the timeline displays the activity of all locations along the entire application runtime, zooming is the method to gain more detailed information. Zooming and scrolling can be executed in the horizontal and vertical dimension to change the shown time interval or subset of locations, respectively. Figure 2.6 shows an example of a global timeline zoomed to a time interval of 0.5 seconds. It is based on a trace of the Weather Research and Forecast Model (WRF) [MDG+04].

Process Timeline Display

Process timelines are similar to global timelines but focus on a single location. The vertical dimension is used to visualize the hierarchy and call dependencies of the code regions by arranging the segments according to their calling depth (see Figure 2.7). Horizontal and vertical zooming and scrolling can be applied in the same way as for global timelines. The process timelines can also be aided by available performance metrics shown as graphs over time. For a detailed comparison of a subset of processing elements multiple process timelines can by displayed simultaneously with aligned time intervals.

Summary Charts

The summary chart displays statistical information about code regions or groups of code regions such as total or average inclusive or exclusive runtime or number of invocations. This information can be gathered from a single location or groups of such and can be shown as pie charts or histograms (see Figure 2.8). While this information is very similar to profiles, the summary charts can be computed for arbitrary time intervals usually selected via a timeline display. Furthermore, all per processing element summaries can be displayed side by side to compare the general behavior of the individual processing elements.

Communication Matrix

The communication matrix shows information to analyze the communication behavior of a measured application such as number of messages, volume, transfer time, and data rates as minimum, average, and maximum values. The values are displayed as two-dimensional matrix with color-coded entries (see Figure 2.9), which allows an easy identification of communication patterns.

Display Arrangement, Performance Radar, and Trace Comparison

All of the above mentioned displays and some more, e.g., for call trees, performance metrics, and location clustering, can be arranged freely within one application window. Figure 2.10 shows an example based on WRF with a clustered process summary chart (i.e., classes of locations with similar behavior are represented only once), a global timeline, and a performance metric display showing the floating point operations per second on the left side, and the function summary, function legend, and call tree on the right side. All these displays are synchronized, thus, whenever the time interval is change within the global timeline, all other displays are re-computed accordingly.

Next to a location local presentation of a performance metric, a color-coded presentation including all locations allows a direct comparison of differences in the behavior of the processing elements regarding a specific performance metric – the so called performance radar. Figure 2.11 shows the floating point operations per second for all locations, which allows in this case an easy identification of compute intensive application phases.

Furthermore, the custom display placement also supports a comparison of multiple traces, which, for instance, is useful for comparing application runs with different parameters or the evolvement of an application in different optimization stages. In addition, approaches for a structural comparison of traces can be applied [WBB12, WMS+13].

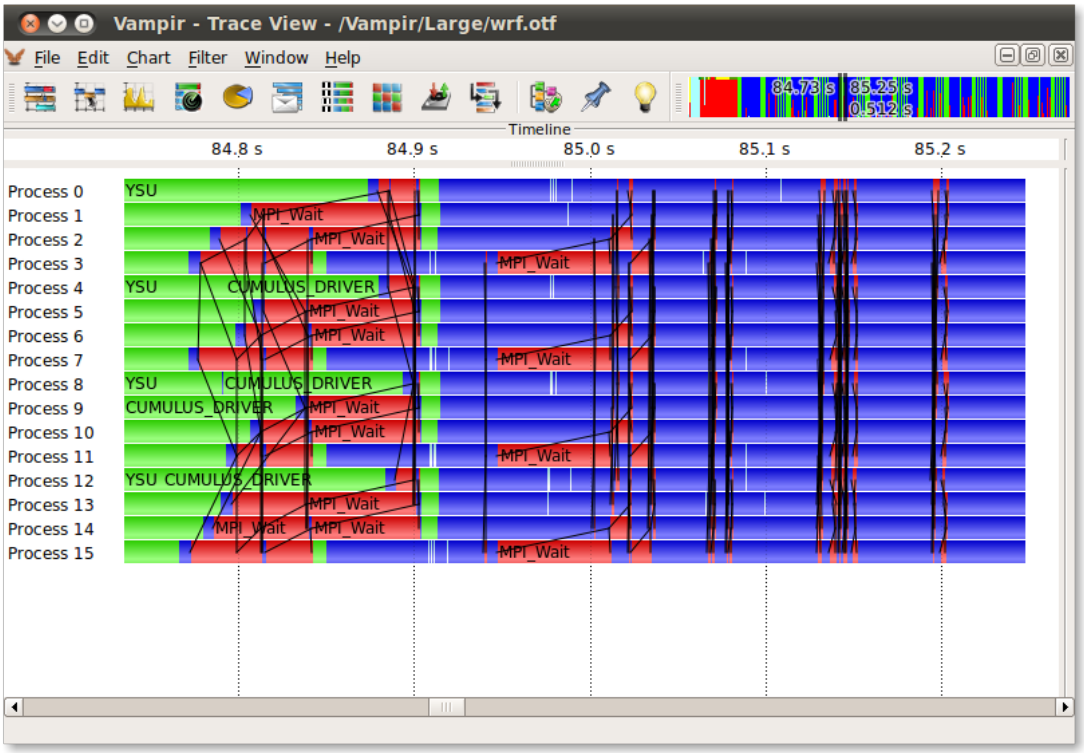


Figure 2.6: Vampir master timeline display showing 16 MPI processes of a WRF measurement zoomed to a time interval of 0.5 seconds (taken from [GWT14]).

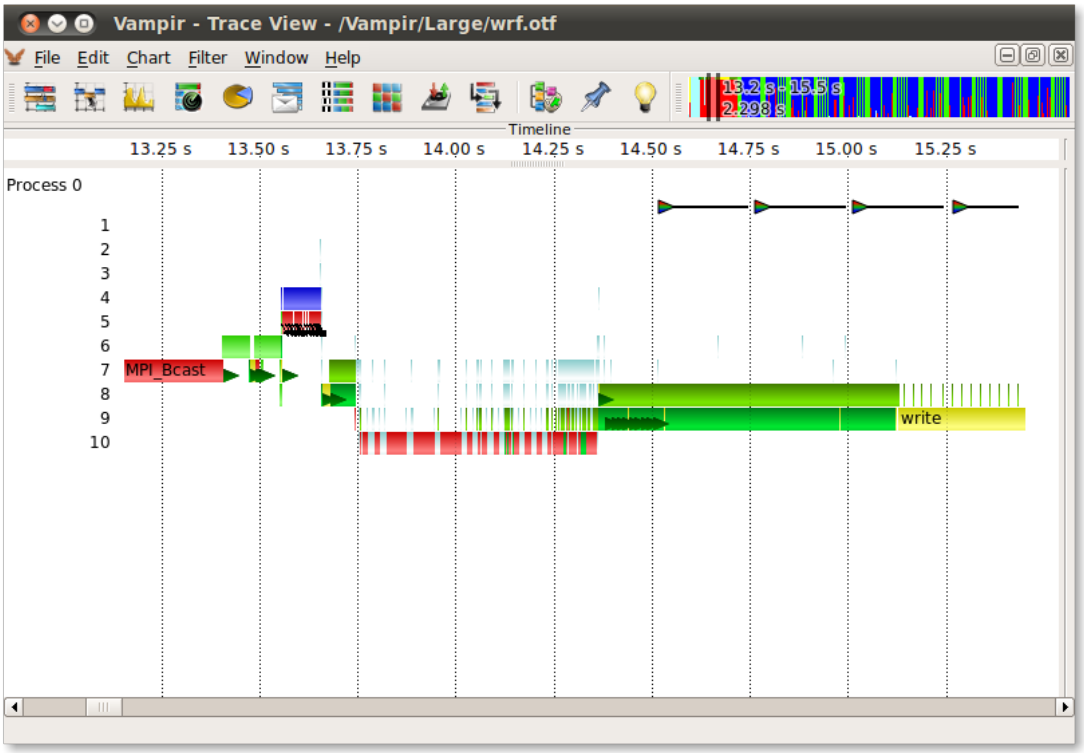


Figure 2.7: Vampir process timeline display showing the call hierarchy of the code regions for process 0 from the example in Figure 2.6 (taken from [GWT14]).

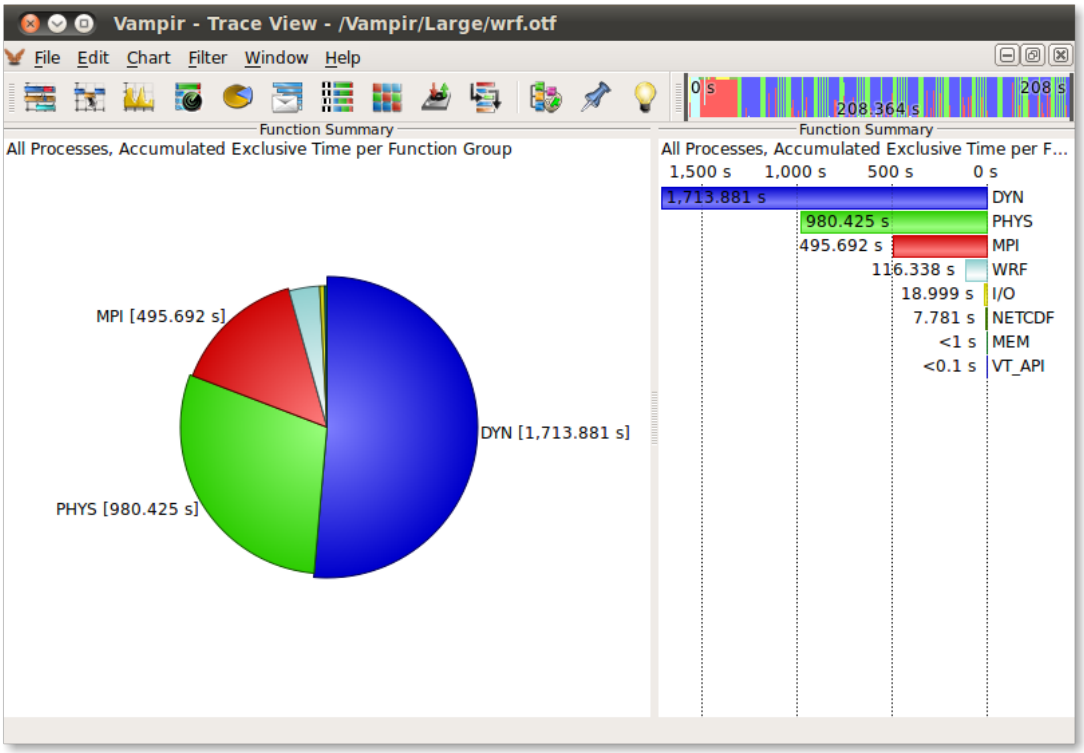


Figure 2.8: Vampir summary chart showing the accumulated exclusive runtime for different function groups as pie chart and histogram (taken from [GWT14]).

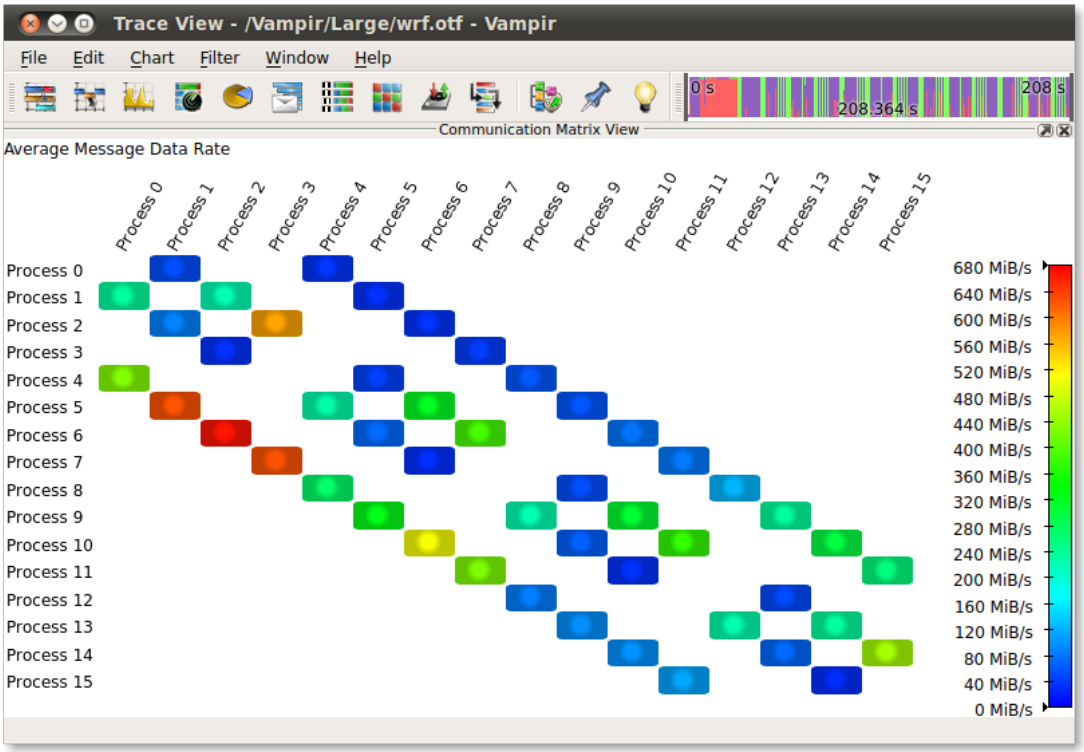


Figure 2.9: Vampir communication matrix (taken from [GWT14]).

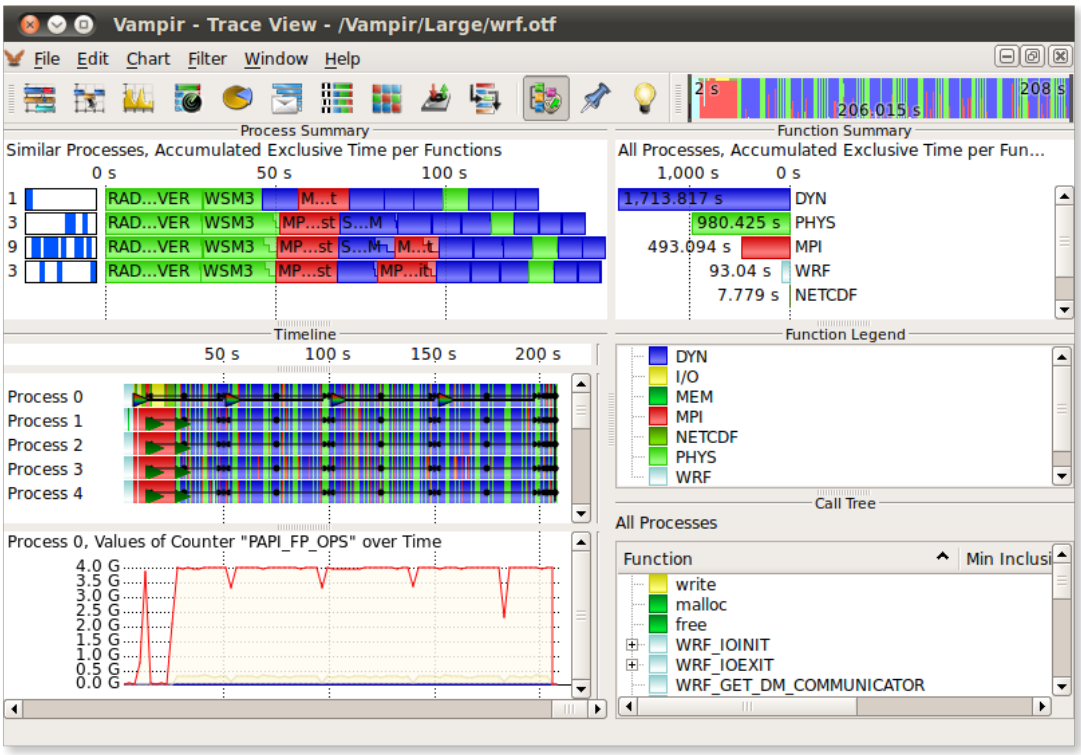


Figure 2.10: Vampir custom display arrangement including a clustered process summary chart, a global timeline, and a performance metric display on the left side, and a function summary, function legend, and call tree on the right side. (taken from [GWT14]).

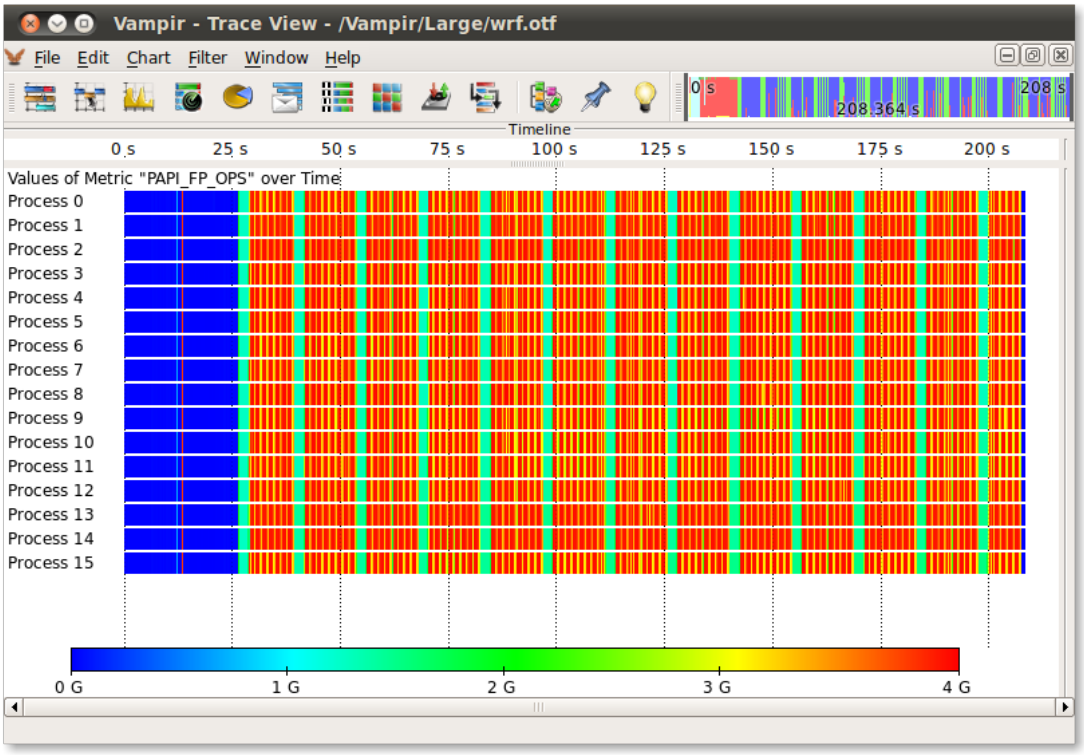


Figure 2.11: Vampir performance radar showing the floating point operations per second, which allows an easy identification of compute intensive phases (taken from [GWT14]).

2.3.2 The Paraver Toolset

The Paraver tool set consists of the Paraver visual performance analyzer [CEP01b] and the Extrae measurement environment [BSC14]. In addition, the Dimemas tool [CEP14] allows trace-based replay to simulate program behavior of a recorded application under alternative conditions such as different CPU speed or different network characteristics. All three tools are developed at the Computer Science devision at the Barcelona Supercomputing Center, Spain since 1991 (partly under different names).
The Extrae monitor supports the recording of all major parallel and accelerator paradigms simultaneously much like VampirTrace. As a unique feature, Extrae records events generated by code instrumentation and sampling probes together. This approach provides additional information between runtime events, which can be useful for long or un-instrumented code regions [BSC14].
The Paraver toolset uses its own trace format, which supports only three basic event record types: states that associate a value for a stream during a time interval, events that represent a punctual event on a stream, and relations that relate two events on two different streams together [CEP01a]. The rather abstract description of these record types originates in their so called semantic free design. Thus, all specific events are mapped to one of these basic record types, e.g., code region enter/leave to states, performance metrics to events, or a point-to-point communications to relations.
The Paraver trace visualizer uses a semantic module to generate a semantic value (numeric value) for each object to be represented, which is a function of time that is computed from the records that correspond to the object. These objects belong to either of two fixed hierarchies: programming model (workload, application, task and thread) and resources (system, node and CPU) [CEP01a]. For both, the semantic value is hierarchically computed according to the general object model structure. Paraver uses three presentation modules: visualization (timelines), textual and statistics (profiles). Each of them handles one of the three types of records [BSC10]. Figures 2.12 – 2.14 show examples of these three presentations.

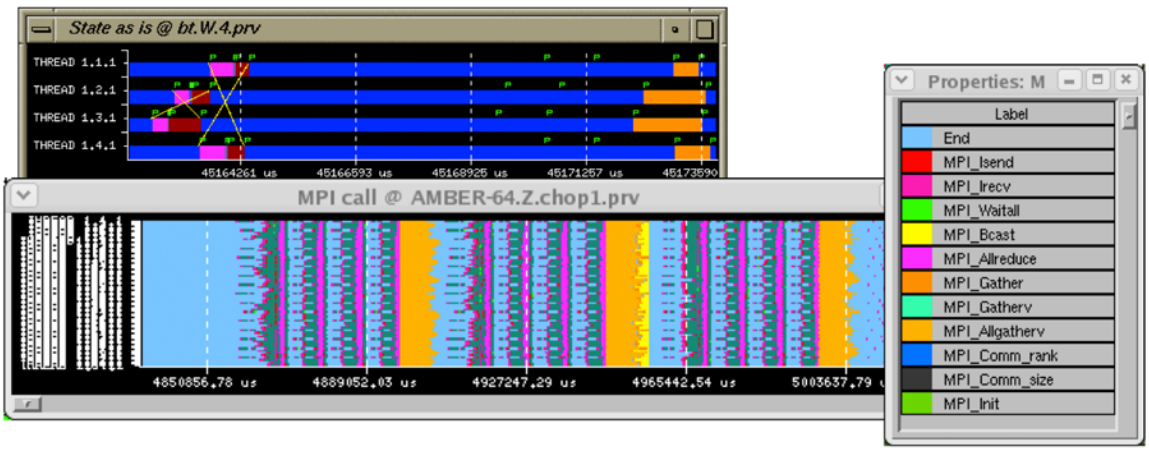


Figure 2.12: Paraver timeline showing phases spent in MPI functions over time (taken from [BSC10]).

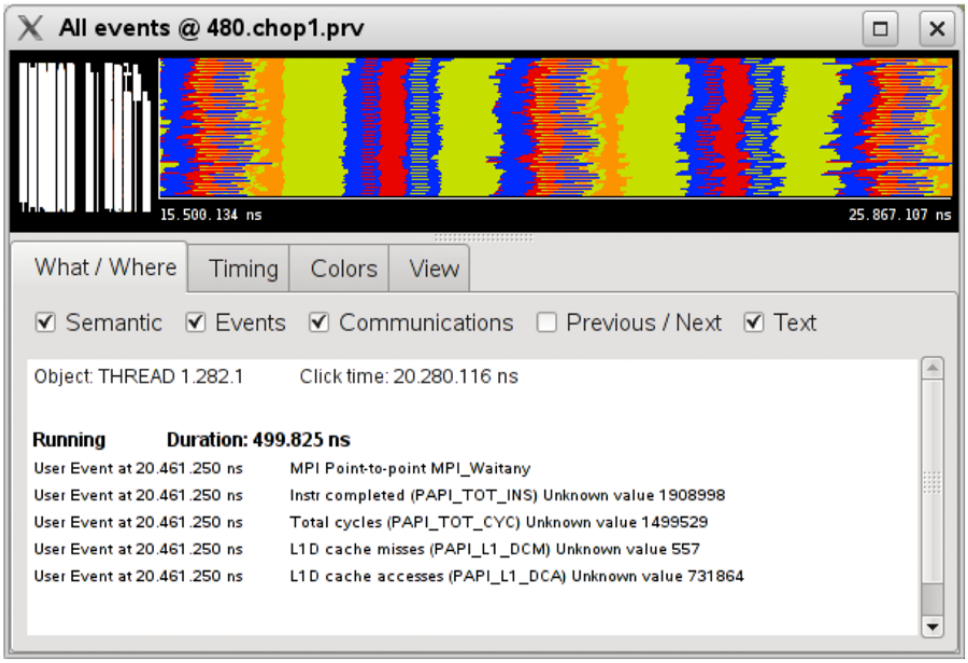


Figure 2.13: Paraver's textual view opens when clicking on a timeline (taken from [BSC10]).

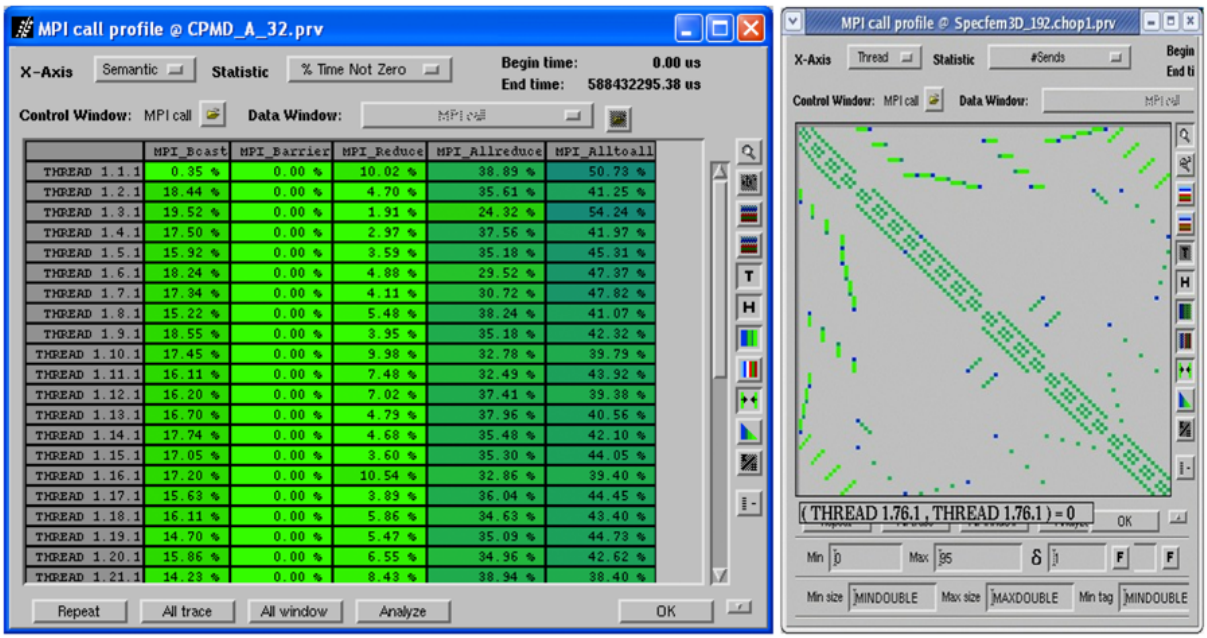


Figure 2.14: Paraver statistics view showing a tabular view of time spent in MPI functions and a communication matrix (taken from [BSC10]).

2.3.3 The Scalasca Toolset

The Scalasca toolset represents a different performance analysis approach than the aforementioned tools. Unlike the performance visualization tools, it applies an automatic analysis to identify patterns of inefficient application behavior. The resulting performance report is presented in a hierarchical viewer called CUBE. Scalasca is developed at the Forschungszentrum Jülich, Germany and the German Research School for Simulation Sciences, Germany. Its predecessor Kojak [WM03] was additionally developed at University of Tennessee in Knoxville, USA. While previous versions included a trace monitor, today, Scalasca uses the unified measurement environment Score-P for trace generation.
The parallel trace analyzer Scout [SDT14a] performs the automatic analysis in Scalasca by searching for predefined patterns of inefficient application behavior. Each identified pattern is given a severity rating ranging from noncritical to critical to allow an easy identification of the most severe performance issues. All predefined patterns are categorized in a hierarchy from general to specific. Typical patterns of inefficient behavior include idle threads or wait times in global or point-to-point communication. Figure 2.15 shows two prominent examples of wait times in point-to-point communication due to unsynchronized messages: the so-called late sender and late receiver pattern. The severity of both patterns is derived from the delay between matching calls, i.e., the longer a communication partner waits the higher the severity. A complete list of performance properties can be found in [SDT14c].

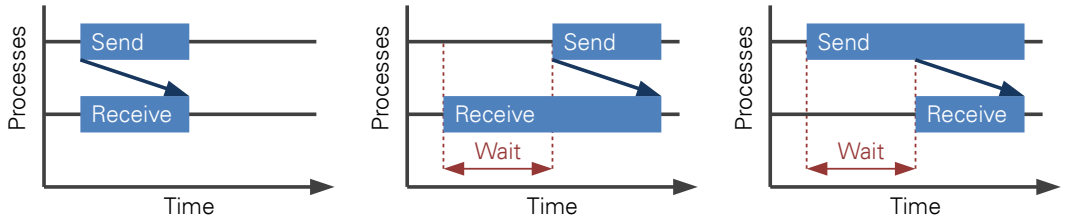


Figure 2.15: Patterns for point-to-point communication: synchronized (left), late sender (center), and late receiver (right), which cause wait times for the sender and receiver, respectively.

The results of the automatic analysis are stored in a single XML file, which is the input for the CUBE display [GSS+12]. It displays the information in three dimensions side by side: a metric dimension, a program dimension, and a system dimension (see Figure 2.16). The metric dimension contains a set of metrics, such as communication time or cache misses that represent the patterns found by Scout. The program dimension shows an application call tree, which includes all call paths onto which metric values can be mapped. The system dimension represents the parallel locations, which can be processes or threads depending on the parallel programming model [SDT14b]. Alternatively, the system dimension can display the performance properties within the system topology (see Figure 2.17).
Each dimension is organized in a hierarchy in the form of weighted trees where the severity rating of each tree node is the aggregation of its sub-trees when collapsed and contains only its own severity when expanded. The severity values are represented by numeric values (absolute or relative), as well as a color-coding, which supports a quick visual perception of the most severe performance properties. Furthermore, the three displayed dimensions are synchronized, so that selecting a sub-tree restricts the other displays to the selected metric. Since the Scalasca approach automatically detects predefined performance inefficiencies, it allows a convenient and low-key identification of typical performance issues.

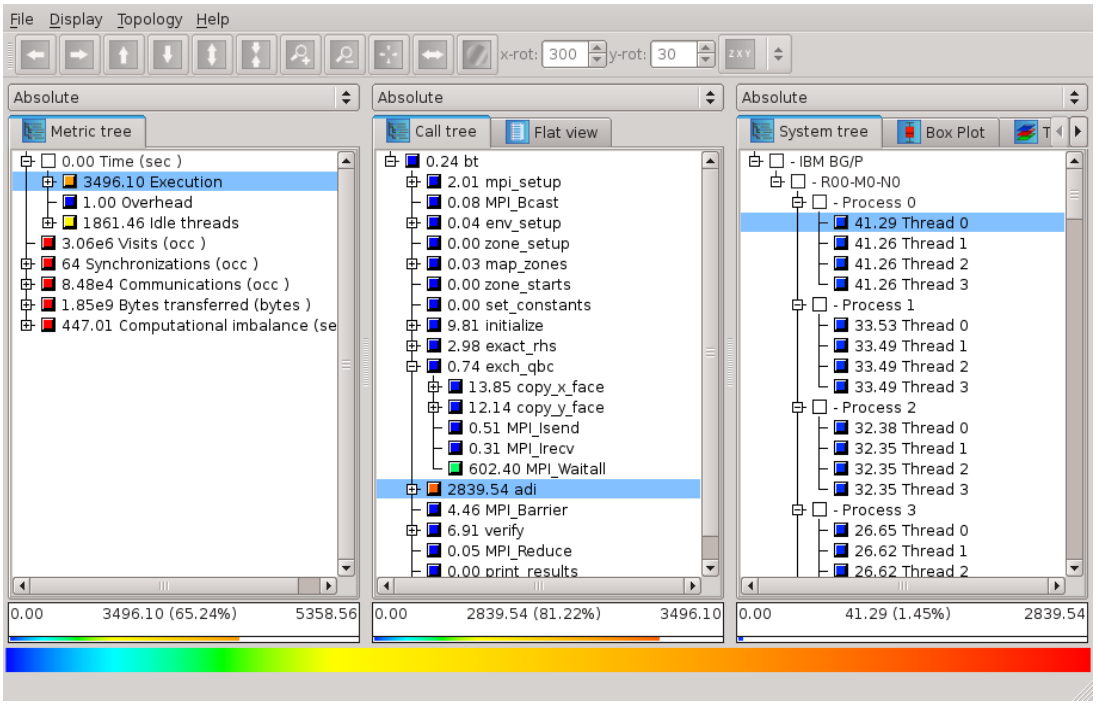


Figure 2.16: Cube display with its three dimensions: metric tree, call tree, and system tree with color-coded severities (taken from [SDT14b]).

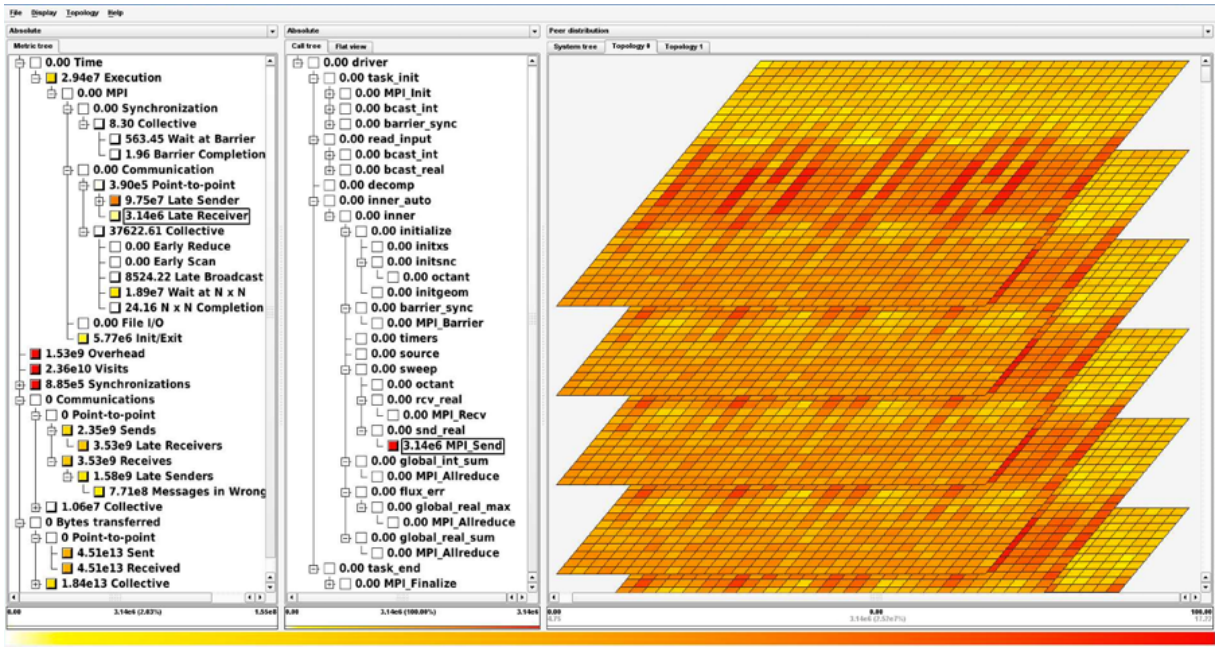


Figure 2.17: Cube display showing the performance metrics mapped to the three-dimensional system topology (taken from [GWW+10]).

2.3.4 The Tuning and Analysis Utilities (TAU)

The Tuning and Analysis Utilities (TAU) provide tools for event-based and sample-based tracing, profiling and profile analysis. For the analysis of event-based traces TAU relies on the aforementioned tools: "We made an early decision in the TAU system to leverage existing trace analysis and visualization tools. However, TAU implements it own trace measurement facility and produces trace files in TAU's own format." [SM06]. Consequently, TAU includes trace file converters to translate TAU traces into formats used by theses tools. However, for large trace files such a conversion might imply a huge effort. Hence, TAU also suggests using Score-P for tracing to generate OTF2 traces [TAU12b] (see Section 2.3.5).
The TAU tracing tools provide two distinct approaches to enhance functionality. The first is PDT [LCM+00], that allows selective source code instrumentation. The second includes early approaches for online performance analysis, one using parallel profiling analysis over MRNet [NMM+08] and the other using file-based online trace analysis with VampirServer [BMSB03]. Furthermore, TAU supports many advanced profiling features such as phase-based and call-graph profiling and its own visualization tool ParaProf. Figures 2.18 and 2.19 show two example visualizations.

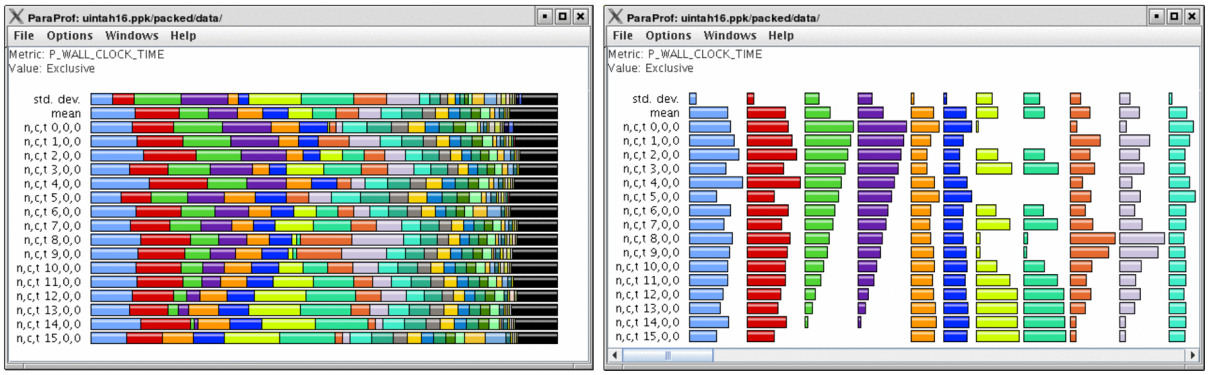


Figure 2.18: ParaProf showing aggregated exclusive runtime per location of each code region. The unstacked bars view (right) allows a comparison of individual code region across location (taken from [TAU12a]).

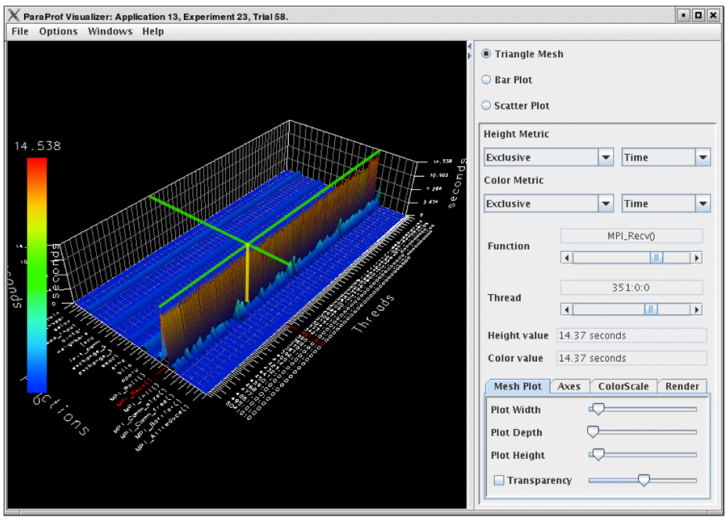


Figure 2.19: ParaProf 3D visualization allows to show two metrics (by height and color) for all code regions and locations (taken from [TAU12a]).

2.3.5 Score-P and the Open Trace Format 2

The previous sections focussed on the analysis functionality of the different tools, which equals the third stage of performance analysis: data representation. Each tool workflow includes its own measurement tool that covers the first and second stage: data capturing and data recording. However, the different measurement environments VampirTrace (Vampir), Extrae (Paraver), Scalasca, and TauTrace (TAU) share very similar techniques for event generation and recording. This led to the idea of joint development and evolution of a unified measurement infrastructure. Today, Score-P [KRM+12] is the joint measurement infrastructure for the analysis tools Vampir, Scalasca, Periscope, and TAU. It comprises the measurement functionality of these tools into a single infrastructure, which provides a maximum of convenience for users next to a reduction of redundant effort in tool development. The Score-P measurement infrastructure allows profiling, event tracing, and online analysis. It contains the code instrumentation functionality supporting various methods and performs the runtime data collection. Figure 2.20 shows an overview of the Score-P architecture and its interfaces to the supported analysis tools.

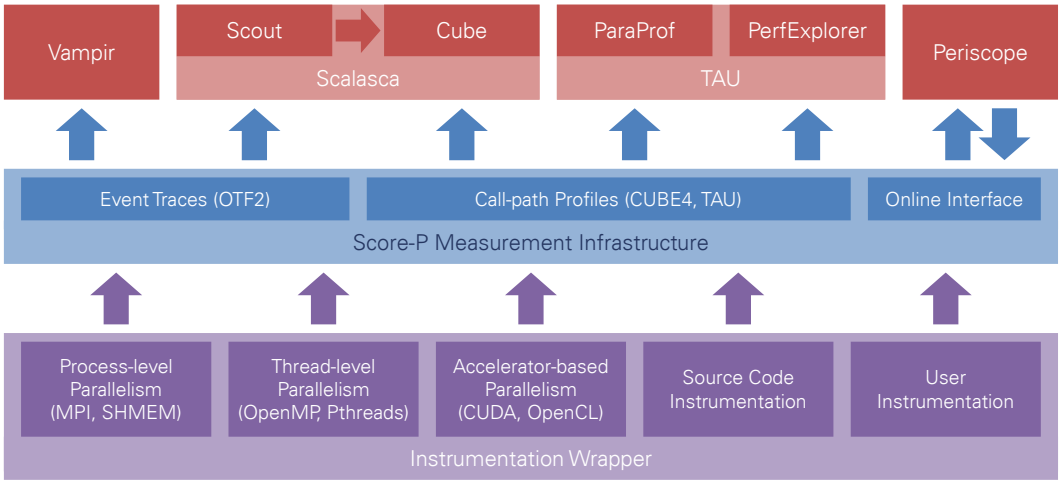


Figure 2.20: Architecture of the Score-P instrumentation and measurement infrastructure and its interfaces to supported analysis tools.

Score-P captures all major paradigms through the following instrumentation techniques:

- Code regions via compiler instrumentation,
- MPI and SHMEM via library interposition,
- OpenMP source code instrumentation using Opari2 [LDTW14],
- Pthread instrumentation via GNU ld library wrapping,
- CUDA, OpenCL instrumentation,
- Selective source code instrumentation via the TAU instrumenter (PDT) [LCM+00],
- Binary instrumentation using Cobi [MLW11], and
- Manual user instrumentation.

While the online interface provides direct access via TCP/IP for online analysis tools such as Periscope, the interface for the other analysis tools is realized by various file formats. For the profiling tools Cube, ParaProf, and PerfExplorer Score-P uses their native formats Cube [SDT14b] and TAU [TAU12a]. Event tracing data is stored within an OTF2 archive, which is covered next.

The Open Trace Format 2 (OTF2)

Event trace data formats of the different tools have many similarities just like the measurement tools themselves [Knu08, EWG+12]. The similarities are also shown in the existence of various converters between most formats [SM06]. Consequently, a unified event trace data format has been developed along with the unified measurement infrastructure. The Open Trace Format 2 (OTF2) [EWG+12] is a highly scalable, memory efficient event trace data format and support library. It is the new standard trace format for Vampir, Scalasca, and TAU. OTF2 is the common successor for the Open Trace Format (OTF) [KBB+06] used by Vampir/VampirTrace and the Epilog trace format [WM04] used by the Scalasca toolset. It preserves the essential features and record types of both and introduces new features such as support for multiple read/write substrates, in-place time stamp manipulation, and on-the-fly token translation. In particular, it avoids copying during unification of parallel event streams.

Since the Open Trace Format 2 is the starting point for the contributions of this thesis, this section presents it in detail as an example similar to many other event trace formats such as the Paraver Trace Format [CEP01a], the TAU trace format [SM06], the Structured Trace Format of the Intel Trace Analyzer [Int14b], as well as its two predecessors OTF and Epilog. In addition, OTF2's integration in multiple analysis tools allows a broad application of the contributions of this thesis (see Section 6).

The Open Trace Format 2 stores all runtime events in the form of trace records, which can be categorized in event records and definition records.

Event Records

Event records mark runtime events and consist of three parts: first a record token that defines the type of an event, second an exact time stamp telling when the runtime event occurred, and third, event specific attributes. They contain all information to entirely reconstruct or replay the application execution behavior. The most common event records are [OTF14]:

- Entering and leaving a code region with an identifier of the code region,
- Sending and receiving an MPI messages storing sender, receiver, communicator, tag, and size,
- Collective MPI operations with the type of collective operation and the communicator,
- Begin and end of OpenMP parallel regions,
- Fork and join of thread teams, and
- Hardware performance metrics with type and value.

Definition Records

Within the event records all references are defined in the form of identifiers, e.g., for the code region in an enter/leave record, or sender and receiver in point-to-point messages. This allows a much higher memory efficiency in comparison to storing names, labels, and descriptions within each event record, in particular, since many of these are referenced repeatedly. The translation of these identifiers is stored within the definition records. The most common definition records are [OTF14]:

- Code regions containing their name, description, and source code location,
- MPI Communicators with their name and communication partners, and
- Hardware performance metrics with a name, description, and measuring unit.

Furthermore, there are definition records that describe the global properties of a measurement like:

- System layout, e.g., a system tree,
- Locations that describe the recorded processing elements,
- The resolution of used time stamps, and
- Various types of groups, e.g., for locations and code regions.

A complete list and a detailed description of all event and definition records can be found in [OTF14].

Read and Write Interface

The Open Trace Format 2 includes a support library that provides interfaces for read and write access to the trace data. The standard access to the trace data is in temporal order, i.e., all events must be written with monotonic increasing time stamps. When reading the trace data, the event information is delivered in the form of user-defined call back handlers in the same temporal order.

All events are separated in event streams that represent exactly one location, which results in one event file per location when the data is stored on the file system. This renders the information on which location an event occurred unnecessary within each single event stream and allows a more efficient storage than mixed-stream formats such as its predecessor OTF. Furthermore, this approach supports an efficient parallel reading of the individual event streams.

Next to the event files for each location, there is a local definition file for each event location that contains mapping tables to convert local identifiers into identifiers within a global scope. Furthermore, there is a global definition file that describes all identifiers within the global scope, i.e., the entire measurement, and the so called anchor file, which serves as an entry point for the OTF2 archive. Figure 2.21 shows the basic file layout of an OTF2 archive.

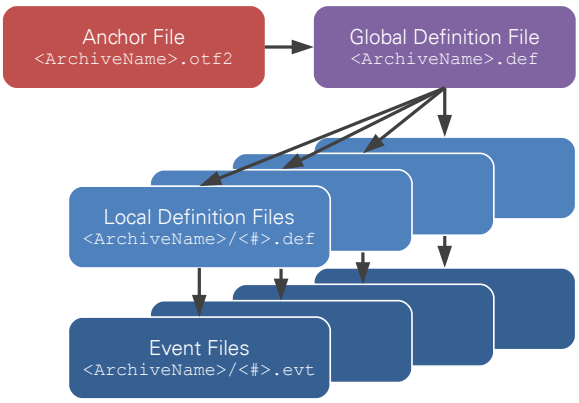


Figure 2.21: Layout of an OTF2 archive containing an anchor file as entry point, one global and multiple local definition files, and event files with records of runtime events.

2.4 Challenges in Event-based Tracing and Related Work

While the previous sections present a general overview on event-based performance analysis, this section focuses on methods related to the concepts of this thesis. This section introduces three urgent challenges in the field of event-based performance analysis and current approaches coping with these challenges. These challenges are, first, scalability in the number of processing elements, second, the management of the enormous data volumes, and, third, the reduction of measurement bias.

2.4.1 Scalability

To help users in the development of scalable software, performance analysis tools themselves must be scalable to the same extend – or even one step ahead. Current HPC systems ranging up to 3 million processing elements [Top14] require software and workflows with extreme levels of concurrency. Monitoring applications at this scale and beyond results in tremendous amounts of event tracing data spread across millions of files; one file for each recorded location. While high-end parallel file systems for such machines are usually equipped to provide enough disk space and sufficient I/O bandwidth, the creation of millions of parallel files is entirely infeasible with all existing parallel file systems [ISC+12]. Requesting more than a few thousand file creations per second affects all jobs and users on a machine. Exact numbers vary on different machines and parallel file systems from 4,000 [ISC+12] to 10,000 [AEH+11] maximum file creations per second.

The reason for this limitation is the handling of file system meta data. Common HPC parallel file systems such as GPFS [SH02] and Lustre [Sun08] focus on increasing data bandwidth, which is primarily achieved by adding more disk drives and more disk controllers. However, large amounts of hardware cannot improve meta data performance since the limiting factors are the number of simultaneous operations and their latency. For massively parallel I/O requests, in particular file creation, their latency has the potential to become the major bottleneck [AEH+11]. In fact, for event tracing tools this meta data wall is one of the most challenging limitations already occurring for systems with tens or hundreds of thousands of cores – not even thinking about systems with several millions of cores arising within the next years [BBC+08].

Two approaches that are dealing with the file system limitations and have been applied to event tracing are SIONlib [FWP09] and the I/O Forwarding Scalability Layer (IOFSL) [ISC+12]. Both approaches merge many logical files into a single or a few physical files. While SIONlib relies on the file system's capability to handle large sparse files to pre-allocate segments for the logical file handles within a single file, the I/O Forwarding and Scalability layer, as the name suggests, provides an I/O forwarding layer to offload I/O requests to dedicated I/O servers that can aggregate and merge requests before passing them to the actual file system.

The IOFSL approach was applied to a full Cray XT5 system measurement with 200,448 cores using the VampirTrace measurement environment [ISC+12]. SIONlib was used in combination with Scalasca to measure 294,912 cores on a BlueGene/P and the aforementioned system [WGM+10].

However, both approaches are currently limited to a subset of systems and in their general applicability. The IOFSL approach requires resources hosting the I/O forwarding servers, which would be best placed on a system's dedicated I/O nodes. But user access to these nodes is usually restricted or impossible. Thus, the I/O forwarding servers can only be deployed on compute nodes reducing the total compute

capability and limiting network and I/O bandwidth. In addition, starting additional server nodes with the application must be supported by the corresponding batch system. SIONlib, on the other hand, requires no server processes but causes unforeseeable synchronization in the parallel application execution, which is critical for event tracing. Furthermore, it depends on MPI for internal coordination, which is infeasible for monitoring non-MPI parallel applications.
Both approaches still cause noticeable overheads for file interaction and have only been demonstrated for small benchmarks with minimal data volumes [ISC+12, WGM+10]. The file interaction overhead in the aforementioned studies was reported with 71 seconds for IOFSL and ten minutes for SIONlib on the BlueGene/P system, which is still very high compared to the small data volumes written. Nevertheless, both approaches achieved a remarkable decrease compared to direct POSIX I/O with a factor of about three for IOFSL and a decrease from 89 to ten minutes for SIONlib.

Both studies demonstrated a drastic improvement over direct file interaction, which pushed the barrier an order of magnitude higher with justifiable overhead. Considering the overhead, the small demonstrated benchmarks, and the aforementioned restrictions, the limitations of event tracing imposed by parallel file systems are still not overcome and remain an unmet challenge.

2.4.2 Data Volumes

As stated before, event tracing can produce enormous amounts of data that need to be handled efficiently. Moreover, trace analysis tools must provide methods to support the user in getting useful information out of this overwhelming amount of data.

Data Volumes in Trace Recording

While the size of a single event record is typically only a few bytes, high event frequencies rapidly generate tremendously huge data volumes. The reviewed applications and application kernels in this thesis (see Section 5.1) show event rates of 50,000 to two million events per seconds per location, which is underlined by other studies [ISC+12]. Given an average event size of 10 bytes this would result in about 0.5 to 20 MiB per second and 300 MiB to 12 GiB for a measurement of 10 minutes. For long-running applications, such as the reviewed Gromacs package [HKS08], a full production run can produce up to 12 TiB per location (see Section 5.4). Of course, this data volumes must be multiplied with the number of recorded locations, which leads to even higher data volumes for large-scale applications. While these data volumes require extraordinary amounts of disk space, for event-based trace recording the data volumes per location are of primary importance since main memory and I/O bandwidth are usually proportional to an increase in core counts.

General Purpose Compression

To cope with large data volumes most tracing approaches use general purpose compression libraries such as the well-establish zlib [DG96] based on the the Lempel-Ziv 77 compression algorithm [ZL77] that provides a good trade-off between compression ratio and overhead [SL11]. While LZ77 compression, as most compression algorithms, depends on the input data, event trace data is typically compressed by a factor of about three to four [WKN12]. Due to the introduced overhead, general purpose compression is not applied on data within the memory buffer but when storing data to the file system.

Encoding Optimizations

In contrast to post-mortem general purpose compression, encoding optimizations of the trace format itself provide a trace size reduction within the memory buffer and the resulting trace file. However, previous enhancements in the encoding reduce trace sizes much less than general purpose compression algorithms [EWG+12]. Still, the discrepancy in size reduction between encoding and general purpose compression hints to unexploited potential in current encoding approaches.

Statistical Clustering

Statistical clustering [NRR97] exploits similarity in parallel application behavior to group processes with similar behavior in a way that all processes within the same group or cluster are more similar to each other than to those in other groups. The data reduction is achieved by keeping only a single representative for each cluster. Consequently, the compression factor for a single cluster equals the number of members. In the case of multiple clusters the total compression depends on the number of members in each cluster and their trace size. The Extrae trace monitor [BSC14] combines statistical clustering [LGS+10] with spectral analysis based on wavelets [LCS+11] to detect iterative patterns within the application, as well.

Pattern Aggregation

Furthermore, there are approaches that use pattern aggregation to reduce trace sizes. These approaches try to detect recurring patterns either within each event stream, e.g., iterations, or across different event streams where, especially, SPMD (single program multiple data) applications contain redundant parallel behavior. Compressed Complete Call Graphs (cCCG) [KN05a, KN05b, KN06] are rooted directed acyclic graphs that combine regular patterns into common sub-trees. The cCCG data structure uses the caller information and time deviations to group leave nodes, i.e., if two calls to the same function vary less than a predefined threshold, they are grouped together. In the same way are all intermediate nodes combined when their caller information matches, they have common sub-trees, and their duration deviates less than the given threshold. Figure 2.22 demonstrates the compression in an Complete Call Graph for a simple example. The proposed approach also presents techniques for optimized cache utilization for the graph node and splitting methods to avoid wide graph nodes [KN06].

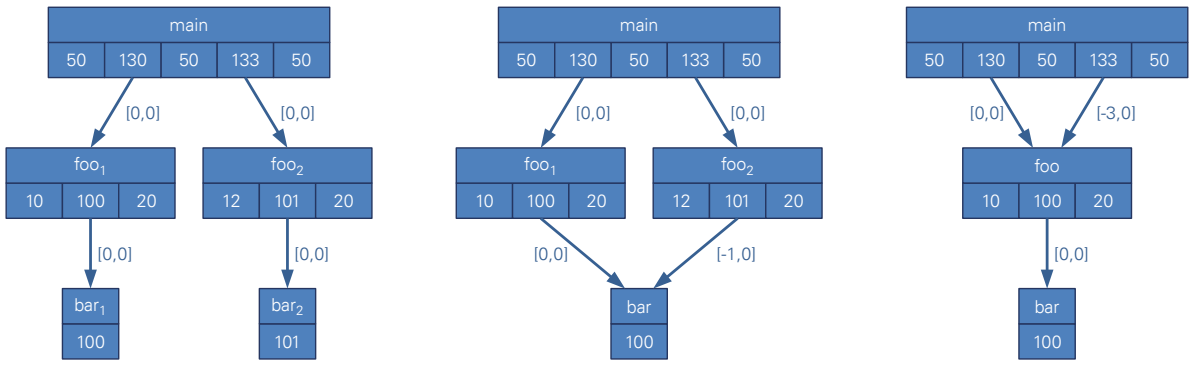


Figure 2.22: Successive compression in a CCG. The first graph (left) shows an uncompressed CCG. Since both calls to function *bar* are compatible they are replaced by a reference (middle). Runtime deviations are propagated to the parent node as a deviation interval [-1, 0]. As both calls to *foo* reference the same child node they are grouped together, as well (right) [KN06].

In addition, a study by Mohror and Karavanic [MK09] evaluates different pattern-based methods for trace compression against several criteria including size reduction and introduced error. The ScalaTrace monitor [NRM+09] presents another pattern-based method explicitly for MPI traces, hence, it does not contain any events for code regions.

Filtering and Selective Instrumentation

Another way to reduce the size of the resulting trace files are manually or automatically applied event filters. A very common filter is the discarding of all function calls when a code region is entered more often than a predefined limit [MKJ+07, GWW+10, KRM+12]. This usually eliminates an overcharge of highly frequent calls to tiny functions, such as helper functions and get/set class methods. Optionally, filters for those functions can be determined beforehand either by statistical source code analysis or a profile run of an application [MLW11]. Selective instrumentation allows to exclude specific functions from being instrumented at all [LCM+00, MLW11]. Furthermore, it is possible to manually define which code regions are instrumented [MKJ+07, KRM+12]. The Paradyn monitor additionally allows to dynamically instrument functions during runtime [BM11] based on their approach for direct binary instrumentation [HMC94].

Data Volumes in Trace Analysis

Event-based trace analysis faces similar challenges with the growing data volumes. Since most tools apply scalable analysis techniques and use either the same amount of resources for analysis as the measured application or an adequate subset [BNM03, GSS+12], the increasing data volumes caused by increasing core counts are a smaller challenge for data processing than the large data volumes per location. However, the increasing data volumes with increasing core counts impose a demanding challenge for the presentation of analysis results. While automatic analysis approaches, such as an detection of root causes of wait states [BGWA10] and determination of the critical path in the application execution [BSG+12], try to pin-point to the most severe performance issues, visual analysis approaches have to circumvent limitations in the screen resolutions as well as in the human perception of information.

The limitations in screen resolution, i.e., presenting more processing elements in a timeline visualization than there are pixels, are passed with an intelligent selection algorithm [NAW+96] or by clustering locations in groups with similar behavior [Bru08, LGS+10]. These approaches allow a reduction of information presented to the user on a complete application representation but when the focus is reduced to parts of the application the detail is kept.

The large data volumes per location are challenging in the data processing as well as the data representation. Since most analysis tools process the recorded events within their own data structures, a trace size reduction achieved by any sort of compression does not reduce the amount of data that needs to be processed and kept in the analysis data structures. On the contrary, the overhead for reading the data is increased by applying an according decompression, except for those approaches that provide options for optimized access to the compressed data structures such as cCCGs [KN06]. Only approaches that reduce trace size by reducing the number of events per location with filters or selective instrumentation allow a reduced effort in trace analysis.

2.4.3 Measurement Bias

To ensure the observed behavior is indeed the real behavior of an application, the measurement process must be as less intrusive as possible – a challenge common to all experimental research.

Measurement Overhead

Measuring an application's behavior always introduces overhead since a software monitor shares resources with the application in one way or another. In event-based performance analysis every runtime event is intercepted, the event is processed, and an event record is stored. For the most common events, entering and leaving a code region, this includes, for instance, the identification of the code region by its name and the code region's context, determine the current time, and store an according event record. During this procedure the application is interrupted in its original execution and, therefore, its behavior is modified. Such measurement overhead can be reduced but never entirely eliminated.

The bias that is introduced by the measurement and how much the original application behavior is altered, depends mainly on three factors: the frequency of events, the interval between two associated events, and the deviation of the introduced overhead, especially, for parallel applications. First, the frequency of runtime events determines the general overhead for a measurement. For very high event rates in the range of millions per second the recorded application can be severely slowed down. Slow downs up to a factor of hundreds have been reported [WDKN14]. While this might render a measurement impossible, in particular, for long running applications and the application behavior is stretched by the slow down factor, the general application behavior and global performance issues are still detectable. Second, fine grained effects might be hidden, especially, when the interval between two associated events is small. For instance, the original runtime of a function call is prolonged by the measurement overhead for processing the according enter and leave. This leads to the effect that the recorded runtime of short-running functions differs considerably from the original runtime. Nonetheless, the relative prolongation is decreasing with an increasing function duration. The third factor, the deviation of the introduced overhead, can have the most biasing effect, in particular, for parallel applications, which is discussed in more detail in the following section.

Intermediate Memory Buffer Flushes

While the overhead for processing the events is usually less deviated by the measurement process, there are two external reasons for significant deviations in the overhead. The first reason are interrupts by the operating system usually referred to as OS noise or OS jitter. The tremendous effects on parallel execution, especially, for large scale applications is demonstrated in [HSL10]. However, this issue affects every part of the system including the measured application itself and is outside a measurement environment's control.

The second reason for significant deviations in the measurement overhead are intermediate memory buffer flushes. To reduce and optimize I/O interaction the recorded runtime events are stored in an internal memory buffer either residing in the monitoring tools or a dedicated tracing library. Whenever such a memory buffer is exhausted, the data is stored at the file system, which causes a severe prolongation of the measurement process of the last recorded event because it is stalled until the file interaction completes. While the prolongation due to such a memory buffer flush can be recorded itself and, there-

fore, considered in the analysis of a single event stream [KRM+12], the parallel behavior is completely disturbed. Since each processing element records different events or at least events with different parameters, e.g., time stamps, such intermediate memory buffer flushes occur entirely uncoordninated. Figure 2.23 demonstrates how an unsynchronized intermediate memory buffer flush can either create or hide a performance issue with the help of the late sender issue (see also Figure 2.15). The same effect can occur for other types of communication patterns and load imbalances.

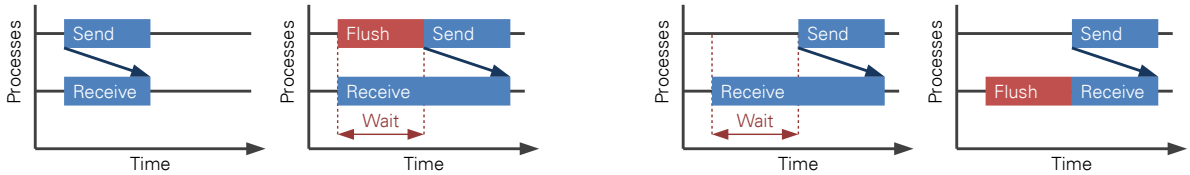


Figure 2.23: Bias on parallel behavior due to intermediate memory buffer flushes. A typical performance issue such as the late sender can be created (left) or hidden (right) when an intermediate buffer flush occurs.

The internal memory buffers are usually rather small, e.g., 10 to 200 MiB [ISC+12], because most of the main memory is left to the application. Hence, such unsynchronized intermediate memory buffer flushes can occur at high frequencies. For instance, an event frequency of one million events per second with a given event size of ten byte would cause a memory buffer of 100 MiB to flush every 10 seconds. For larger memory buffers the frequency of memory buffer flushes would decrease, however, the duration of the interrupt would increase. In any case, due to the aforementioned impact on the interaction of processing elements, the entire measurement starting from the first initiated memory buffer flush must be considered incorrect. This assumption is also supported by the case study in Section 5.4. Consequently, for typical memory buffer sizes most of the recorded application behavior must be considered incorrect, which renders a feasible recording of long running application impossible.

The only way to eliminate the bias of uncoordinated intermediate memory buffer flushes is by eliminating the memory buffer flushes themselves. In this sense, only a drastic reduction of memory requirements during runtime would avoid such bias and, therefore, allow the recording of long running applications. However, most of the methods for trace size reduction presented above, in particular, all methods for huge trace size reductions such as general purpose compression and cCCGs, introduce too much overhead themselves to be applied during runtime. Therefore, this challenge must be considered as unsolved.

2.5 Open Challenges and In-memory Event Tracing

Currently, in event-based tracing all three challenges must still be considered unsolved: First, the number of resulting trace files limits scalability; primarily due to the meta data wall in parallel file systems [AEH+11]. There exist two approaches (SIONlib [FWP09] and IOFSL [ISC+12]) that circumvent the limits emerging at scales higher than a few thousand cores and allow tracing up to hundreds of thousands of cores. However, both approaches are currently limited to a subset of systems and applications, introduce still noticeable overheads and, moreover, it is not clear whether these approaches will also be sufficient for exa-scale systems with tens or even hundreds of millions of cores.
The enormous data volumes created constitute the second challenge. All existing approaches targeting the huge data volumes including the application of general purpose compression [DG96], optimized encoding [EWG+12], statistical clustering [NRR97, LGS+10, LCS+11], and pattern aggregation [KN06, MK09, NRM+09], turn out to be either of too small effect or introduce remarkable overheads, thus, they can be only applied for a post-mortem trace size reduction but not during runtime.
The third challenge is the introduced measurement bias, in particular, due to uncoordinated intermediate memory buffer flushes. With their disruptive character, they can falsify the recorded program behavior and either create or conceal critical performance issues. Consequently, the entire measurement starting from the first initiated memory buffer flush must be considered incorrect, which renders a feasible recording of long running application impossible.

These three unresolved challenges in event tracing provide the motivation and target of this thesis. It proposes concepts for an in-memory event tracing workflow, that increases memory efficiency and dynamically adapts trace size during runtime to keep performance data of an entire measurement within a single fixed-size memory buffer. Such an in-memory event tracing workflow meets all three previous challenges: First, it not only overcomes the scalability limitations due to the number of resulting trace files but eliminates the overhead of file system interaction all together. Second, the enhanced encoding techniques and event reduction lead to remarkable smaller trace sizes. Finally, an in-memory event tracing workflow completely avoids intermediate memory buffer flushes, which minimizes measurement bias and allows a meaningful performance analysis.

2.6 Summary

With High Performance Computing systems getting more and more powerful but also more and more complex the demands for developers of parallel applications have risen considerably. Developing applications that utilize the enormous capabilities of these complex systems requires a continuous process of optimization – a task nearly unfeasible without the help of appropriate supporting tools.
Performance analysis tools assist developers not only in identifying performance issues in their applications but also in understanding their behavior on complex heterogeneous systems. Tools gather information about the behavior of an application during runtime by either recording runtime events or by periodically sampling the current state. While sampling approaches rely on their sampling frequency to gain information about an application, event-based monitoring records information when specific runtime events occur. Sampling or event-based generated information can be either aggregated to summarized

information about different performance metrics (profiling) or stored individually by keeping the precise time stamp and further specific metrics for each event (tracing). Profiling with its nature of summarization decreases the amount of data that needs to be stored during runtime. However, profiles may lack critical information and hide dynamically occurring effects. In contrast, event tracing records each event of a parallel application in detail. Thus, it allows capturing the dynamic interaction between thousands of concurrent processing elements and enables the identification of outliers from regular behavior – with the cost of huge generated data volumes.
For the analysis of event-based traces there exist two primary approaches. First, the visualization of the recorded event data in the form of timelines, which display performance metrics such as the active code region over time on the horizontal axis and the locations on the vertical axis. Such a visualization is usually assisted by various other information such as summaries, a context menu, and a communication matrix. Two prominent tools with this approach are Vampir and Paraver. The second approach is an automatic analysis of the recorded traces based on the search for predefined performance issues such as unsynchronized communication or load imbalances. Scalasca applies such an analysis and represents the gathered information in the Cube display in a metric context, call tree context, and system context.
The fact that all major analysis tools have measurement tools with similar functionality led to the idea of joint development and evolution of a unified measurement infrastructure. Today, Score-P serves as the joint measurement infrastructure for the analysis tools Vampir, Scalasca, Persicope, and TAU. Score-P comprises the measurement functionality of these tools into a single tool, which provides a maximum of convenience for users next to a reduction of redundant effort in tool development. Furthermore, the Open Trace Format 2, a unified event trace data format and support library was developed along with the unified measurement infrastructure. The Open Trace Format 2 serves as the reference and target for the contributions of this thesis, which allows a broad applicability of the enhancements.

In event-based tracing three challenges arise that must still be considered as unsolved. First, the increase in core counts demands highly scalable tools and workflows for event-based tracing. In particular, the number of resulting event trace files is already pushing against the limits of parallel files systems. While there exist two approaches that circumvent the limits emerging at scales higher than a few thousand cores and allow tracing up to hundreds of thousands of cores, both approaches are currently limited to a subset of systems and applications, introduce still noticeable overheads and, moreover, it is not clear whether these approaches will also be sufficient for exa-scale systems with tens or even hundreds of millions of cores. The enormous data volumes created constitute the second challenge. All existing approaches targeting the huge data volumes including the application of general purpose compression, optimized encoding, statistical clustering, and pattern aggregation, turn out to be either of too small effect or introduce remarkable overheads, thus, they can be only applied for a post-mortem trace size reduction but not during runtime. The third challenge is the introduced measurement bias, in particular, due to uncoordinated intermediate memory buffer flushes. With their disruptive character, they can falsify the recorded program behavior and either create or conceal critical performance issues. Consequently, the entire measurement starting from the first initiated memory buffer flush must be considered incorrect, which renders a feasible recording of long running application impossible.

The following two chapters, the main part of this thesis, present concepts for an in-memory event tracing workflow that provides approaches to meet all three challenges.

The next chapter introduces methods to keep an entire measurement within one fixed-sized memory buffer and forms the first main part of the contribution of this thesis. Such an in-memory event tracing workflow meets the first challenge, by not only overcoming the limitations of current parallel file systems but eliminating the overhead of file system interaction all together. It uses a highly enhanced encoding combined with low overhead event reduction strategies to dynamically adapt trace size during runtime to the given memory allocation. This ultimately results in remarkably smaller trace sizes, which is the second challenge. Furthermore, an in-memory event tracing workflow completely avoids intermediate memory buffer flushes and, therefore, minimizes measurement bias, the third challenge, and allows the tracing of long running applications.

After that, Chapter 4 introduces the Hierarchical Memory Buffer, which is the second part of the contribution of this thesis. The Hierarchical Memory Buffer is a new data structure that uses hierarchy information such as calling depth or event class to presort events according to these hierarchy attributes. Hence, the Hierarchical Memory Buffer allows to perform the aforementioned event reduction operations with minimal overhead. In addition, several typical analysis requests can benefit from a hierarchy-aided traversal of recorded event data.

3 Concepts for In-memory Event Tracing

This chapter specifies prerequisites for an in-memory event tracing workflow and defines three key steps to keep an entire measurement within a single fixed-size memory buffer. The three key steps filtering, enhanced encoding techniques, and event reduction are discussed in detail.

3.1 In-memory Event Tracing

To clearly characterize the target and contribution of this thesis and to avoid misconceptions, this section defines the term *in-memory event tracing* as it is used in the context of this thesis. Following the classification of Section 2.1 *event tracing* is the process of recording runtime events and storing them individually in a trace for the purpose of performance analysis. In this sense, event tracing covers data capturing and data recording, which leads to a trace containing the recorded events. The applied data representation (profile, timeline, or automatic analysis) is not covered in this process.
In-memory describes the location of the recorded event traces, which is a CPU's main memory including its registers and cache hierarchy. This implies that there is no interaction with the file system during the event tracing process. In this context, the further processing after the end of an application measurement is left undefined. The resulting event trace within memory can either be kept in main memory and directly forwarded to an analysis component (see Chapter 6) or stored at the file system for future analysis. It is important that there is no file system interaction during the measurement, which can critically affect the measurement of the application behavior. Both parts combined can be expressed as follows:

> *In-memory event tracing describes a method in performance analysis where runtime events are recorded and stored individually in a trace that remains in main memory for the entire measurement workflow.*

Consequently, the main challenge for an in-memory event tracing workflow is to keep events of an entire measurement within a single fixed-sized memory buffer. Most event tracing approaches buffer recorded events in main memory to reduce file system interaction. Once the memory buffer is exhausted, the measurement is either aborted or event data is flushed to a file. An in-memory event tracing workflow, however, must avoid both of the above. Therefore, an in-memory workflow must apply methods to reduce the amount of data in the memory buffer by either reducing the number of events or the amount of memory per event.
Additional constraints are that, first, these methods introduce minimal overhead to avoid additional measurement perturbation, second, the measurement can contain an arbitrary but finite amount of events, and, third, the memory buffer can be of arbitrary but fixed size. In other words, keeping an event trace of arbitrary size within main memory cannot be achieved by increasing the size of the memory buffer accordingly. Furthermore, the size of the memory buffer is usually small (about one to ten percent of main memory) since most of main memory is left to the observed application to minimize measurement

bias. Otherwise, an application that runs out of memory due to a too large memory buffer would render the measurement useless. In addtion, in the context of this thesis it is agreed that each location of a parallel application is recorded in its individual event trace (which combined represent the complete parallel program), i.e., each location has its own memory buffer. Its obvious that it is possible to create constraints for a measurement where the resulting event trace is completely useless, for instance, when the memory buffer is too small to contain any events. Therefore, in the context of this thesis it is further agreed that memory buffer sizes and other measurement parameters remain within typical boundaries.
At any rate, there is no universal method to realize an in-memory event tracing workflow. However, this thesis will demonstrate that a combination of established, as well as new methods, allows to keep a measurement of arbitrary size within a single fixed-sized memory buffer – the main criterion for in-memory event tracing. The established and new methods can be categorized in three major steps within an in-memory event tracing workflow: *selection and filtering*, *encoding and compression*, and *event reduction* (see Figure 3.1).

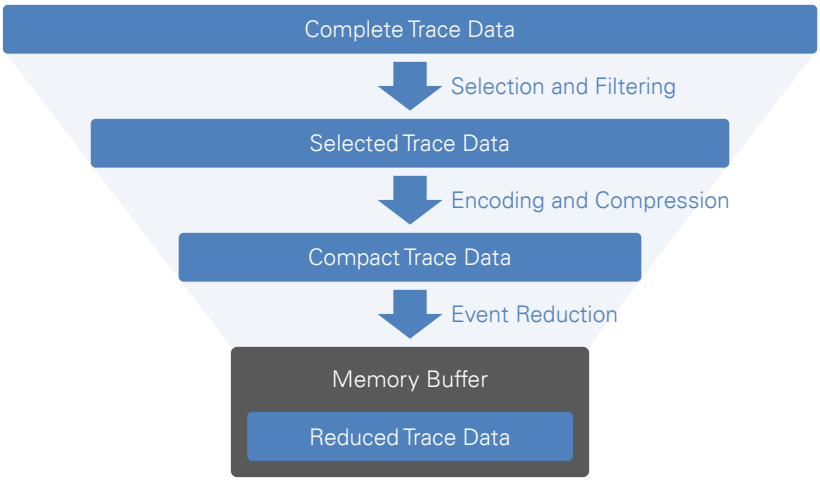


Figure 3.1: The three main steps to allow in-memory event tracing: selection and filtering, encoding and compression, and event reduction.

The first step contains methods to either select events for monitoring before the measurement starts or filter events during runtime and is discussed in Section 3.2. This step contains mainly existing and well-established methods presented in Chapter 2 and, additionally, optimizations gained from the Hierarchical Memory Buffer data structure. The second step is an efficient storage of event tracing data within the memory buffer. This requires a compact encoding and low-overhead compression and is detailed in Section 3.3. This step builds on existing encoding methods and presents new enhanced techniques to remarkably increase memory efficiency. While these first two steps can significantly reduce memory allocation, they fail the by far most important criterion for an in-memory workflow: they cannot guarantee that the data of an arbitrary measurement fits into a single memory buffer of fixed size. Consequently, the third and last step is entirely different. It is triggered whenever the memory buffer is exhausted; typically this is the point where the memory buffer is either flushed to a file or the measurement is aborted. The crucial point is making memory space available again by reducing the events already stored within the memory buffer while in the same time introducing minimal overhead. This step covers a completely novel approach and is discussed in detail in Section 3.4.

3.2 Selection and Filtering

In order to realize an in-memory event tracing workflow, methods for selection and filtering form the first step. Both can be applied during instrumentation or during runtime and can refer to single events, classes of events, and program phases. Typical techniques include:

1. *Selective phase instrumentation*: Restrict instrumentation to specific program phases, e.g., a phase that is of specific interest or where a previous analysis detected a performance issue.
2. *Selective event instrumentation*: Restrict instrumentation to specific events, e.g., exclude code regions that are not of interest.
3. *Manual selective instrumentation*: Users explicitly define application phases or code regions that will be recorded.
4. *Static phase filtering*: Phases of an application are statically filtered based on predefined criteria, e.g., in iterative codes only every n-th iteration is recorded.
5. *Dynamic phase filtering*: Phases of an application are dynamically filtered based on the runtime behavior of the application, e.g., in iterative codes only those iterations are recorded, that deviate from average behavior.
6. *Static event filtering*: Events are filtered based on static predefined criteria, e.g., code regions are filtered after they have been called a certain amount of times.
7. *Dynamic event filtering*: Events are dynamically filtered based on the runtime behavior of the application, e.g., calls to code regions are filtered that are shorter than a minimum duration.

The first three techniques are applied before or during the instrumentation step of an application. Some compilers, e.g., the GNU compiler, allow steering of the instrumentation via plugins. In addition, there are tools that support selective instrumentation, such as PDT [LCM+00] and Cobi [MLW11]. The big advantage of these first three techniques is, that all phases and events excluded from instrumentation are not recorded at all, which not only reduces the amount of stored data but also the measurement overhead. However, these techniques require specific knowledge or information about the application. Such information can be provided by the user, by a previous analysis or can be automatically generated. Two approaches to automatically generate filters for selective instrumentation are static code analysis [MLW11] or the comparison of symbols in an instrumented and un-instrumented binary to find those functions that would be inlined without compiler instrumentation[1] [WDKN14]. The latter follows the assumption that tiny functions, e.g., helper functions or get and set class methods, are usually heavily called but contribute only little to the overall application behavior.

In contrast, the methods four to seven are applied during application runtime. Static and dynamic phase filtering methods require the identification of iterative behavior in an application. Based on that, some tools provide a so called rewind functionality [MKJ+07, KRM+12]. Thereby, the beginning of an iteration is marked with a rewind point in the memory buffer. At the end of each iteration the memory buffer can be rewound and all events beginning from the last rewind point are discarded [WDKN14]. This can be done either statically to keep, for instance, every n-th iteration or dynamically based on predefined criteria, e.g., keep only representatives of each class of iteration.

[1] Automatic compiler instrumentation typically disables the inlining of very short functions.

Filtering single events can be done immediately when recorded or delayed after they have already been stored in the memory buffer. Removing events immediately supports only static filtering, in the sense, that before the event occurs the filter condition is already set. For instance, when filtering functions after they have been called a certain amount of times, the number of calls to a function is already known before the function is entered again and, therefore, the decision whether the next enter and leave events of that function are filtered is already made. Many tools support such filters in the form of user defined filter rules that can exclude functions or function groups (e.g., all functions starting with a certain prefix) entirely or after they have been called a certain amount of times [MKJ+07, GWW+10, KRM+12]. In contrast, removing functions based on their duration requires the removal of events already stored in the memory buffer because the duration is first known when the function is left, i.e., the time the leave event is recorded. At this point, the corresponding enter event is already stored in the memory buffer. Such dynamic filters are usually more complex and introduce to much overhead and, therefore, are not applied in any existing tools.

At any rate, this thesis focuses on encoding (Step 2) and event reduction (Step 3) rather than selection and filtering (Step 1), which is mainly covered in the related work chapter. The primary contribution of this thesis, to the first step of selection and filtering, is allowing to filter events during runtime more efficiently. In particular, the Hierarchical Memory Buffer allows to apply dynamic event filtering (Method 7) very efficiently because of its hierarchy-aided layout. This way, new filter techniques can be incorporated that have not been possible before because either the hierarchy information is not available or they introduce too much overhead. Exemplarily, Section 3.4.4 presents a method for runtime filtering of function calls based on their actual duration.

3.3 Enhanced Encoding Techniques

The second step in an in-memory event tracing workflow includes methods for efficient encoding and compression. This step builds on existing methods in the Open Trace Format 2 [EWG+12], which is similar to many other event trace formats, such as the Paraver Trace Format [CEP01a], the TAU trace format [SM06], the Structured Trace Format of the Intel Trace Analyzer [Int14b], as well as its two predecessors OTF and Epilog. Starting from OTF2, this section describes new encoding methods that can noticeably increase memory efficiency of the event trace data format. As stated in Section 2.4.2 there is a huge gap between the memory efficiency of general purpose compression and event-centric encoding, that these new encoding techniques target to minimize.

Naturally, the topic of efficient encoding of a data format is fairly technical and specific to each format. However, this section discusses general concepts that can be transferred to other event trace formats because most event trace formats share a basic design and have many similarities. In addition, the integration of the Open Trace Format 2 in multiple analysis tools allows a broad application of the new concepts of this thesis (see Chapter 6).

While there exist other approaches in storing event records (e.g., as aligned C data structures in VampirTrace [MKJ+07]), a binary encoding of event trace records has proofed to be most efficient in terms of runtime overhead as well as memory overhead [EWG+12, WKN12]. A detailed comparison of the different event trace formats in Section 5.2 supports this statement, as well.

3.3.1 Binary Event Representation

At first, this section describes the basic memory representation of an event record in a binary encoding. An event record consists of three main parts: First, a record token that defines the type of an event, e.g., entering or leaving a code region, sending or receiving a message; second, an exact time stamp telling when the event occurred; and third, event specific attributes, e.g., a region ID for a region enter record. Figure 3.2 shows a generic event record with two attributes in the basic record design of the Open Trace Format 2. This is similar to others such as the Epilog trace format, which contains only an additional leading byte that stores the length of the record [WM04].

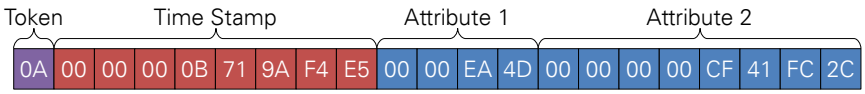


Figure 3.2: Basic memory representation of event records. First, a one-byte record token defines the type of an event, followed by a time stamp of eight bytes telling when the event occurred; and third, event specific attributes, one with four bytes and one with eight bytes.

The following exemplary event sequence demonstrates the basic memory representation of event records in the binary encoding in more detail. The example is a call to the function *MPI_Send* to send a message to another location in the message passing implementation MPI [MPI12]. This results in an enter event for entering the function, a message event for the sending of the message, and a leave event for leaving the function. In addition, an arbitrary performance metric, e.g., cache misses, is recorded with each entry and exit of a function. The timing information for the message event is matched either to the enter or leave event for the according function call.

Figure 3.3 represents the memory representation of such an event sequence in a binary encoding. All event records are stored contiguously in the memory buffer but to ease reading they are placed below each other in the figure. This event sequence and the generic event record in Figure 3.2 are used throughout this section to illustrate the different encoding techniques.

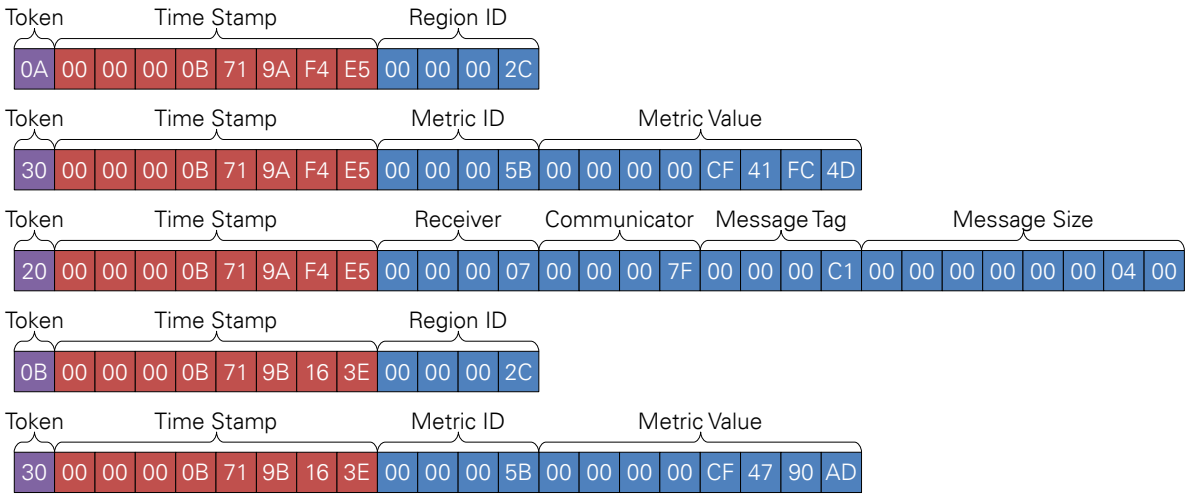


Figure 3.3: Memory representation of an exemplary event sequence: Each event record starts with a record token identifying the event type, followed by a time stamp and event specific attributes.

3.3.2 Splitting of Timing Information and Event Data

Usually, trace formats store events and their according timing information in a single event record like in Figure 3.2. This is consequential to the general idea of event tracing because an event is only meaningful with information about the time it occurred. However, the advantage of storing timing information and event data separately is the elimination of redundant timing information for consecutive events with identical time stamps. In that case, the timing information becomes a separate event record with its own token (see Figure 3.4). This time stamp record sets the timing information for all following events and is valid until the next time stamp record. The management of such separately stored timing information can be handled transparently by the read and write library.

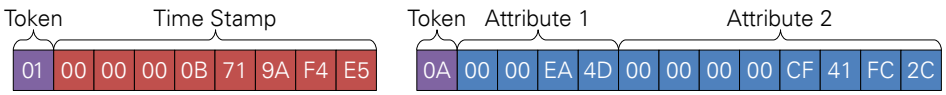


Figure 3.4: Splitting of timing information and event data results in two separate records: First, a record storing only the time stamp that is valid until the next time stamp record; and, second, a record storing only the event attributes.

The advantage of this methods becomes clearly visible when applied to the example event sequence. Figure 3.5 shows that instead of five time stamps only two time stamps are stored. This leads to a reduction of 24 bytes for redundant timing information. The trade-off is an additional byte for the token of the new time stamp record.

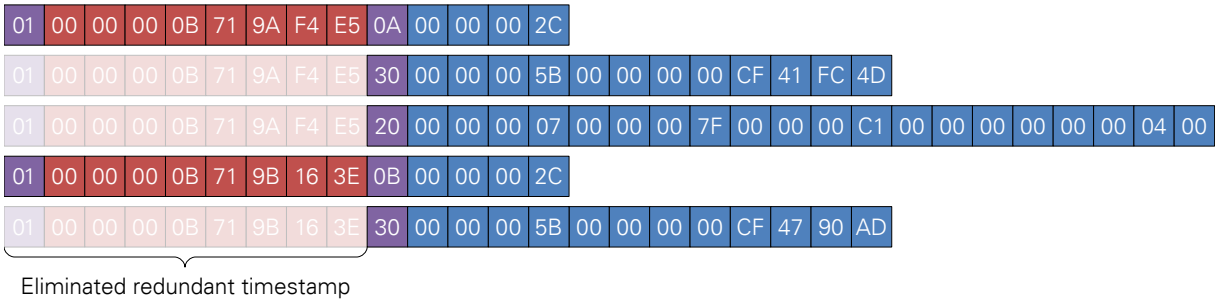


Figure 3.5: Splitting of timing information and event data for the example: Instead of five time stamps only two time stamps are stored; which leads to a reduction of 24 bytes for redundant timing information.

3.3.3 Leading Zero Elimination

The memory reserved for each attribute of an event record is usually determined by the largest value the element theoretically represents. In this way, a region ID is typically stored as a 32-bit integer while a hardware performance counter is stored as a 64-bit integer. However, the majority of values is much smaller than the hypothetical maximum but results in the same memory allocation. For integer values, which are most of the values, this frequently results in a number of leading zero bytes for Big Endian encoding. For Little Endian encoding the effect is the same; but with trailing zero bytes. The following examples use Big Endian encoding to ease reading. At any rate, omitting these leading zero bytes can reduce the resulting memory allocation for the vast majority of event records. To still be able to read this not fix-length representation of the value, the number of remaining data bytes is stored in front of the value. Figure 3.6 demonstrates this method for the generic event record. The splitting of timing information and event data, as well as the leading zero elimination, have already been integrated in current versions of the Open Trace Format 2 [EWG+12].

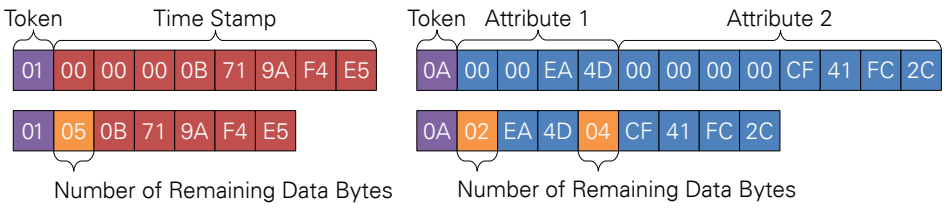


Figure 3.6: Leading zero elimination: Omitting all leading zero bytes. The number of remaining data bytes is stored in front of them.

3.3.4 Delta Encoding

In general, there are two different types of values that are stored: monotonic increasing values like time stamps and arbitrary values like region IDs. Furthermore, some of the monotonic increasing values begin with a very high offset such as time stamps and hardware performance counters. In this case, storing only the difference (delta) to the previous value leads to much smaller values to store. In combination with the described leading zero elimination this results in less memory allocation for the stored value. Figure 3.7 depicts this effect for the exemplary event sequence.

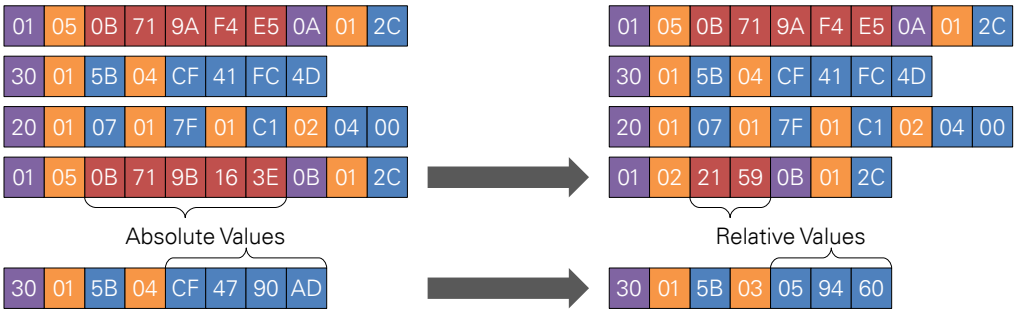


Figure 3.7: Delta encoding for the exemplary event sequence: Since time stamps and hardware performance counters are monotonic increasing only deltas are stored.

3.3.5 Event Distribution and Encoding Implications

The previous optimizations can be enhanced even more by encoding very small numbers directly into the token byte. With the described leading zero elimination a token is always followed by a very small number: The number of remaining data bytes for the first attribute. This number can be encoded into the token byte by adding the number of remaining data bytes of the first attribute to the token (see Figure 3.8). This eliminates the additional byte storing the number of remaining data bytes for each event record.

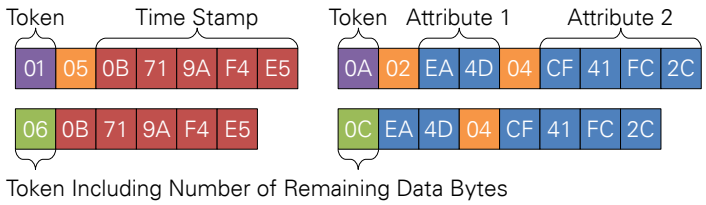


Figure 3.8: Merging of token and number of remaining data bytes of the first attribute.

However, the merge of token and number of remaining data bytes must be considered in the distribution of the tokens since each token and, therefore, also the merged tokens, must uniquely identify the event type. This leads to multiple tokens for each event type that depend on the size of the first attribute, e.g., five different tokens for a 32-bit integer and nine tokens for 64-bit integer; one for each possible number of remaining data bytes. This includes an additional token indicating that the number of remaining data bytes is zero in case the value itself is zero. Since the token byte can only represent 256 unique states, it is obvious that this method cannot be applied to every event type. Therefore, it is important to identify those events that are most common in event traces.

Encoding for Most Frequent Events

To determine the events that are most common a set of real-life applications and application kernels was surveyed. The applications include a subset of the SPEC MPI 2007 benchmarks [MWL+07], the NAS Parallel Benchmarks [BLBS92] in version 3.3, as well as the real-life applications Gromacs [HKS08], COSMO-SPECS+FD4 [LGW+12], and Semtex [BS04]. They represent real-life applications or extracted kernels from various research fields using HPC systems. A detailed description of the individual applications and kernels can be found in Section 5.1.

Figure 3.9 shows the distribution of all event classes by their number of occurrences in the traces of the reviewed applications and kernels. It reveals that the dominant event records are time stamps, code region enter and leave events, and metrics. Thereby, the number of metric events depends on the number of different metrics and the frequency they are recorded. The SPEC MPI benchmarks include one metric event on every region entry and exit. Hence, the number of metric events equals the number of region enter/leave events. In the measurements of the NAS Parallel Benchmarks metrics are gathered with a fixed frequency of 100 μs. Thus, the ratio of metric events depends on the application's general event frequency. To cover all three possible scenarios for the inclusion of metric events, the three real-life applications do not contain any metrics. If multiple metrics are recorded the share increases accordingly and they can easily become the dominant event class. The share of MPI point-to-point events is for all

applications either small or even insignificant. MPI collectives and all other events only have a negligible share. Of course, this ratio is based on completely instrumented and recorded event traces and changes accordingly when methods for selection and filtering as in Section 3.2 are applied.

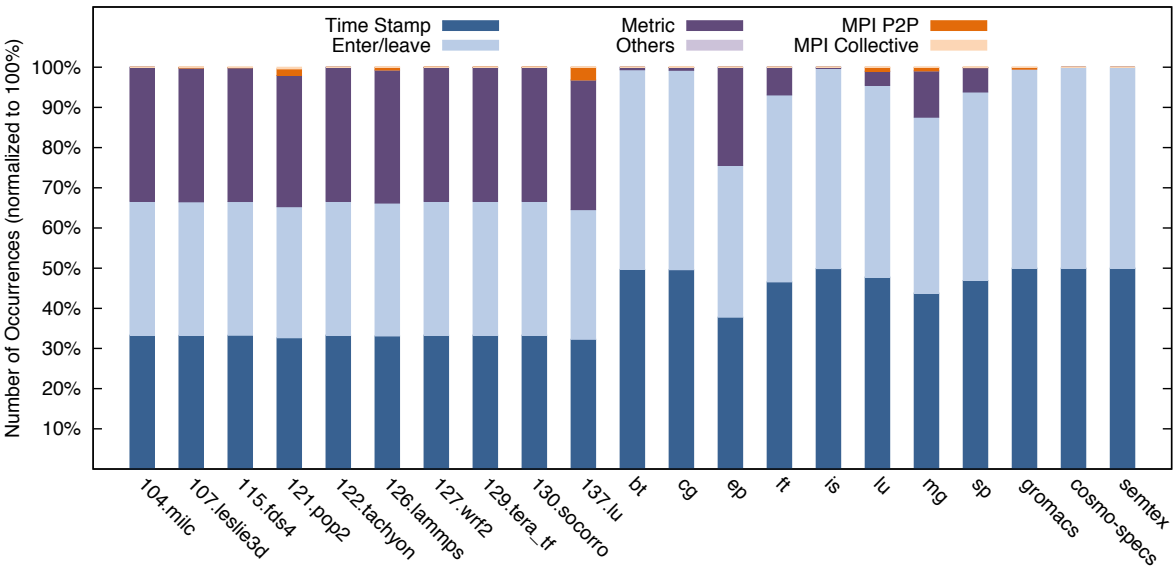


Figure 3.9: Distribution of event classes by number of occurrences for the SPEC MPI 2007 benchmarks *104-137*, the NAS Parallel Benchmarks *bt-sp*, as well as the real-life applications Gromacs, COSMO-SPECS+FD4, and Semtex.

From Figure 3.9 it can be inferred that the optimization described above is most useful for time stamp, enter/leave, and metric events. In this respect, 20 additional tokens are required, which is feasible to apply. Figure 3.10 shows this method for the example event sequence.

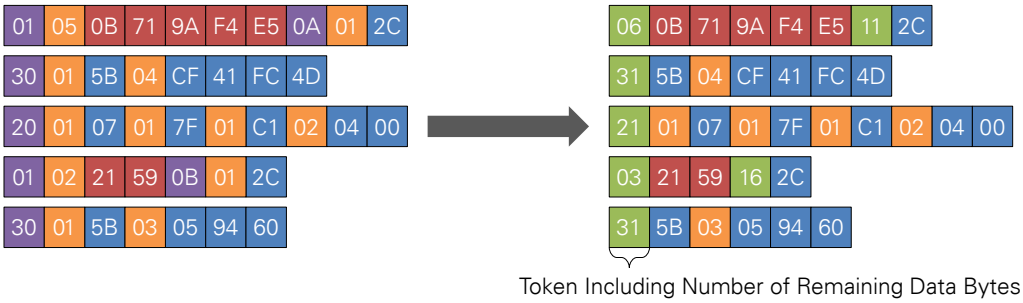


Figure 3.10: Merging of token and number of remaining data bytes of the first attribute.

Encoding of Enter and Leave Events

Still, a lot of tokens remain unused. For instance, the Open Trace Format 2 version 1.3 uses 63 different tokens including internal steering tokens. Even with the additional 20 tokens almost three quarters of the token space remains idle. Since enter and leave events are of the most common events and contain only a single attribute whose value is usually small, they are perfect for extra optimization. A further improvement upon the previous technique utilizes the remaining token space for the encoding of small

region IDs directly into the token byte of enter event records. This way, about 150 of the smallest region IDs can be covered in the token ID. The region ID in leave records can be omitted entirely since they can be obtained by keeping a function call stack when reading the trace, which can be handled transparently by the read and write support library. Figure 3.11 illustrates the effect for the example event sequence.

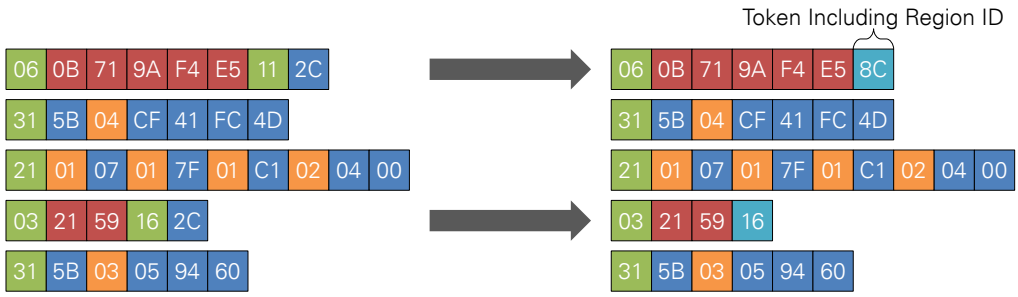


Figure 3.11: Merging of token and region ID for enter events; the region ID for leave events is omitted.

The efficiency of this techniques depends on the distribution of the region IDs, since the potential of this methods can only be utilized when the small region IDs are also the most frequently used region IDs; otherwise, the effect is irrelevant. Figure 3.12 shows that most of the surveyed applications and kernels only use a very small number of region IDs in total. Thus, the majority of region IDs can be encoded directly into the token. This means, that for the surveyed applications the size of most enter events and all leave events is reduced to one single byte. However, whether or not the small region IDs are actually the most frequent region IDs, cannot be determined by the measurement represented in Figure 3.12. But Section 5.2 evaluates the effect of this method in detail.

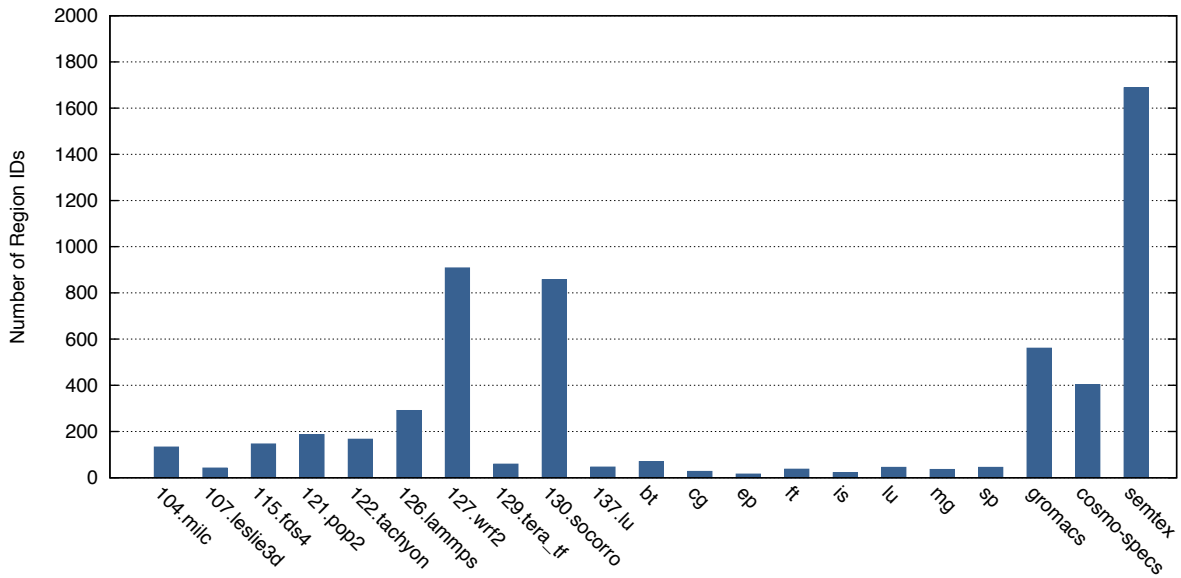


Figure 3.12: Number of different region IDs for the SPEC MPI 2007 benchmarks *104-137*, the NAS Parallel Benchmarks *bt-sp*, as well as the real-life applications Gromacs, COSMO-SPECS+FD4, and Semtex.

3.3.6 Timer Resolution Reduction

The last techniques target to reduce of the size of another frequent event record, time stamps. Typically, event tracing tools check for available timer sources and use the one with the highest resolution, e.g., a CPU's cycle counter, which is in the order of nanoseconds. However, events are recorded at a much lower frequency. Figure 3.13 shows that for the reviewed applications and kernels the event frequency is in the order microseconds, which is also supported by other studies [ISC+12]. In addition, the remaining error after post-mortem synchronization of non-global timer sources is usually in the range of microseconds [DKMN08]. Therefore, it is possible to reduce the stored timer resolution of high frequency timer sources without perceivably degrading the accuracy of the timing information. An optimal reduction factor, in the sense that it is the highest possible reduction before the program behavior is in any way modified, can only be determined correctly after the measurement is completed. During the measurement a reduction factor can only be obtained by heuristics, mainly based on the resolution of the timer source and usual error rates. Thus, only a mild reduction can be applied to ensure a safety buffer. Furthermore, in a technical view, the reduction should be applied as a bit shift rather than a division. In this respect, the reduction factor is restricted to powers of two.

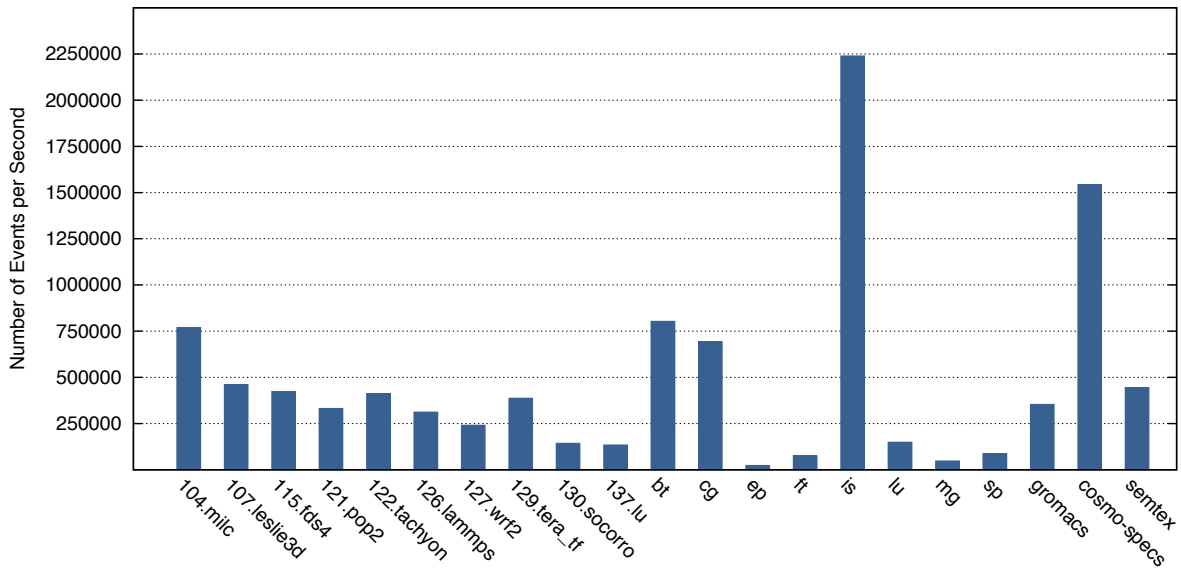


Figure 3.13: Average event rates of the reviewed applications and kernels.

In contrast to all other techniques the timer resolution reduction applies lossy encoding even though the loss of accuracy is rather small with a mild reduction of the original timer resolution. Therefore, Section 5.2 provides a quantification of the resulting error and compares loss-less (without timer resolution reduction) and lossy encoding separately.

This section presents enhanced encoding techniques that allow a remarkable increase in memory efficiency of the event tracing format without increasing runtime overhead of the tracing library. Section 5.2 evaluates these techniques in detail in terms of memory allocation and runtime overhead and compares the enhanced encoding techniques to other event trace formats, as well as general purpose compression.

3.4 Event Reduction

The third and final step in an in-memory event tracing workflow provides methods for event reduction. While the first two steps, selection and filtering plus encoding and compression, can achieve a remarkable reduction of the stored data they lack the capability to reduce the data to a fixed size – the size of the memory buffer. The methods of both steps have an upper bound of their reduction potential, i.e., if the input data is large enough they cannot keep the memory buffer from overflowing. In other words, they fail the by far most important criterion for an in-memory workflow: they cannot guarantee that event data of an arbitrary measurement fits into a single memory buffer of fixed size. Without a guarantee to keep the event data within a single memory buffer, however, an in-memory event tracing workflow is impossible.

Consequently, event reduction follows a completely different approach than the methods of the first two steps. In the first step, events or program phases are either deselected during instrumentation or filtered during runtime while, in the second step, an enhanced encoding and compression allows an efficient storage of the remaining events. In contrast, the third step, event reduction, is triggered only when the memory buffer is exhausted; typically this is the point where the memory buffer is either flushed to a file or the measurement is aborted. The crucial point is making memory space available again by reducing the number of events already stored within the memory buffer while at the same time introducing minimal overhead.

Each event reduction operation selects events by dynamic criteria and discards them from the memory buffer. Such a selection follows similar heuristics as the selection and filter methods in the first step; the main difference is that events matching a criterion are not filtered in any case but only when a reduction is inevitable, i.e., the memory buffer is exhausted. In addition, the criteria for event reduction are not static but adapt to previous event reduction operations. In other words, if all events matching a specific criterion have already been reduced, the next criterion is chosen for event reduction. This allows an incremental reduction of events. This effect will become more clear, when the individual reduction operations are discussed in detail. Furthermore, all information for a reduction criterion must be computed directly from a single or a few events to enable a fast determination whether or not an event is discarded without requiring the complete context.

This section introduces four strategies for event reduction that meet these requirements: a reduction by the order of occurrence of the events, by their event class, by the current calling depth, and by the duration of a code region. It is obvious, that each of these four methods has individual advantages and disadvantages. The main focus of the comparison of these methods is based on two criteria. First, the quality of the remaining information. Since performance analysis has two major goals, to better understand an application's behavior and to identify potential performance issues, the comparison is based on how good these goals can still be achieved with the reduced event set: Is it still possible to understand the behavior of the application and is it still possible to detect occurring performance issues? Second, the granularity of the individual event reduction operation. In this respect, granularity means the amount of data that is discarded in a single event reduction operation. If the reduction steps are too large, a lot of information might unnecessarily be discarded.

3.4.1 Reduction by Order of Occurrence

The first strategy is to reduce events by their order of occurrence. This means that events are either discarded or kept depending on the time they occurred. If the memory buffer is capable to store n events, there are three different ways this method can be applied:

1. Store the first n events, i.e., recording is stopped once the memory buffer is exhausted.
2. Store the last n events. This method requires a cyclic buffer that starts overwriting events in the front of the buffer whenever the end of the buffer is reached.
3. Store either the first or last n events within a specific application phase.

Since events are always in temporal order, these methods provide events for a certain time interval $[t_1, t_2]$ with $t_{start} \leq t_1 \leq t_2 \leq t_{end}$ and t_{start}, t_{end} are the starting and end point of the application or the application phase (third method), respectively. If the measurement is short enough, i.e., there occur less events than the buffer is able to record, the interval $[t_1, t_2]$ equals $[t_{start}, t_{end}]$. Otherwise, either $t_2 < t_{end}$ or $t_{start} < t_1$ applies. How much smaller the interval $[t_1, t_2]$ is in comparison to the complete application interval depends on how much events need to be discarded in order to keep the remaining events within a single memory buffer. If there is a total of m events for the complete measurement and the memory buffer is capable of recording n events, than the recorded interval represents the fraction of $\frac{n}{m}$ of the entire application, given a fairly equal distribution of event sizes.
These methods provide the complete application behavior within the recorded interval $[t_1, t_2]$; either at the beginning, at the end, or somewhere in the middle of an application, depending on which method is chosen. Outside of this application interval, i.e., in the intervals $[t_{start}, t_1)$ and $(t_2, t_{end}]$, there is no information about the application's behavior available because all according events are discarded. Thus, a performance analysis based on these methods allows a good understanding about the recorded interval of the application but cannot provide any information about the part that was discarded. The same applies for the ability to detect performance issues. Performance analysis can detect performance issues that occur within the recorded application interval but performance issues occurring outside of this interval cannot be detected. In addition, an analysis might also miss performance issues on the border of the recorded interval since there is only partial information available.
Therefore, the quality of the overall information about an application strongly depends on the structure of the application and whether the right interval is selected for recording. Since it is not possible to determine beforehand, where performance issues might occur – because this will be the result after a performance analysis – in most cases this strategy may deliver poor results in terms of the information that can be obtained about the application.
Nevertheless, the reduction by the order of occurrence has a very high granularity. Methods with a fixed starting point enable an event-wise reduction because recording can be stopped at any event. A cyclic memory buffer for methods with a fixed end point might overwrite multiple events since they have different sizes in the memory buffer. Still, the granularity is in the order of a few events.
An event reduction by the order of occurrence is the most basic of the four event reduction strategies. In particular, the first method with a fixed starting point is not too different from a measurement abortion but with the difference that all recorded events are kept for analysis instead of being dismissed. However, due to their very high granularity and their easy application these methods serve well as a fallback if all other event reduction operations fail.

3.4.2 Reduction by Event Class

The single events that are recorded can be categorized into different classes of events, e.g., entering and leaving a code region, point-to-point or collective communication, performance metrics like hardware performance counters, or I/O operations. Naturally, not all of these different event classes are of same importance when analyzing an application. For instance, for an analysis of the communication behavior, obviously, communication events are very important while specific hardware performance counters, like cache misses, are less important. For an analysis of single thread performance it is the other way around. Hence, it is possible to order the different event classes and start event reduction with the least important event class. Since an automated heuristic to order the event classes by importance can only provide very vague guesses this order should be specified by the user depending on the focus of their analysis. If none is given, an automatic order can be based on the typical distribution of memory requirements of each event class and start reduction with the one that uses most of the memory. Figure 3.14 shows such a distribution of event classes by size of memory allocation for the reviewed applications and kernels.

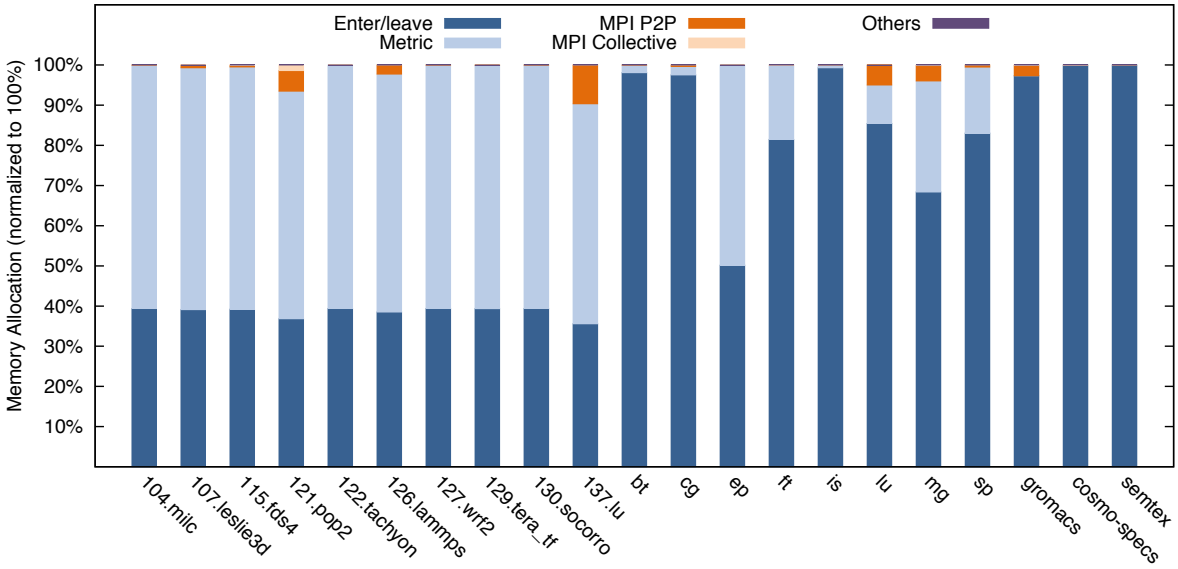


Figure 3.14: Distribution of event classes by size of memory allocation.

In contrast to the first strategy that provides complete information within the recorded interval, this approach provides information for the entire application interval. However, the information is restricted to the events of those event classes that remain in the trace. For this strategy it is important that the event classes are represented by disjunct subsets N_i of the set M containing all events occurring in an application, with $i = 1..j$ and j the number of different event classes. In addition, each event must be an element of exactly one subset N_i. In other words, each event belongs to exactly one event class. Again, if all events fit into the memory buffer the set of remaining events $\bigcup_{i=1..j} N_i$ equals M; otherwise $\bigcup_{i=1..k} N_i$ is a true subset of M, with $k < j$ being the number of remaining event classes.
Hence, the remaining events allow a partial performance analysis. Analyzing the remaining events results in a good understanding of those aspects of the application that are represented by the remaining event classes. For instance, if communication events are available a complete communication analysis is possible. About application behavior deducible by events of discarded event classes, no knowledge

can be obtained at all. The detection of performance problems shares the same restrictions. Performance issues that can be recognized by the remaining event classes can be fully analyzed, whereas performance issues deducible by reduced event classes cannot be detected. Furthermore, performance issues that can only be derived by a combination of multiple event classes cannot be detected when one of these event classes has been discarded. In this respect, the quality of the overall information about an application's behavior mainly depends on an appropriate order of event classes in terms of their importance.
The granularity of this reduction operation relies on the distribution of the different event classes by their memory allocation. Unfortunately, the statistical survey of the reviewed applications and kernels shown in Figure 3.14 reveals that there are only three dominating event classes: enter/leave events, performance metrics, and point-to-point communication. All other events have only a marginal fraction of the total memory allocation. This means that the granularity of the reduction steps is very low, which limits the potential of this event reduction strategy. Nonetheless, this strategy can serve quite well to sort out events of one or two of the main event classes if they are of less importance.

3.4.3 Reduction by Calling Depth

Next to an order by event class, events can also be ordered by their calling depth. The third event reduction strategy uses this order and starts reduction with those events on the deepest call stack level which implies that no further events on this or a deeper call stack level are recorded. This strategy is based on the assumption that events on the deepest call stack level usually contribute less to the overall understanding of the application behavior than those on higher levels. Still, these events may be the source for a performance issue.
This strategy allows a partial performance analysis for the entire application interval similar to an event reduction by event class. In this case, the individual call stack levels form the disjunct subsets N_i of the set M containing all events occurring in an application, with $i = 1..j$ and j the maximum calling depth. In the same way, each event must be an element of exactly one subset N_i or, in other words, each event can be assigned to exactly one call stack level. In addition, a meaningful analysis requires that all events that form natural pairs, such as enter and leave of each function call, are assigned to the same call stack level. If all events fit into the memory buffer the set of remaining events $\bigcup_{i=1..j} N_i$ equals M; otherwise $\bigcup_{i=1..k} N_i$ is a true subset of M, with $k < j$ the number of remaining call stack levels.

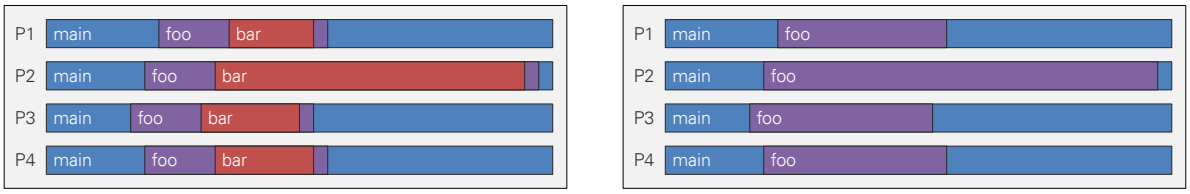


Figure 3.15: The correlation between cause and impact in a basic timeline visualization. The complete even trace (left) shows a load imbalance caused by the function *bar* on process two. When the call stack level that contains *bar* is discarded, its impact is still visible in *foo*.

Similar to the second strategy, the behavior and potential performance issues can only be fully reconstructed with the events in the remaining call stack levels. However, while the first two strategies completely discard the information with the events that carry them, this strategy allows to obtain parts of the information from higher call stack levels. In particular, when reducing the call stack level that contains

the events that mark a performance issue, the actual cause of the performance issue is lost. Yet, a performance analysis might still allow to recognize the impact of this performance issue in the remaining call stack levels. Figure 3.15 demonstrates this correlation between cause and impact for a simple load imbalance in a basic timeline visualization. The complete event trace clearly shows a load imbalance caused by the function *bar* on process two. When the event reduction by call stack level engages, the deepest call stack level containing *bar* is reduced and the cause of the performance issue cannot be identified anymore. However, the impact of the performance issue and, therefore, the performance issue itself is still detectable. Hence, the knowledge gained about an application's behavior and the ability to detect performance issue is reduced but not completely lost for individual aspects. Of course, the impact of the performance issue becomes more and more blurred when further call stack levels are discarded. In the example load imbalance, the performance issue is completely lost if the second deepest call stack level, containing *foo*, is eliminated, as well.

The second criteria, the granularity of single reduction steps, of this strategy strongly depends on an application's structure with regard to its call stack level distribution. An ideal case is a deep and equally distributed call stack, which allows a reduction in very fine grained steps. Figure 3.16 shows the call stack distribution in an ideal example and for some selected applications and application kernels. In addition, Figure 3.17 presents the call stack distribution for all SPEC MPI 2007 and NAS Parallel Benchmarks applications.

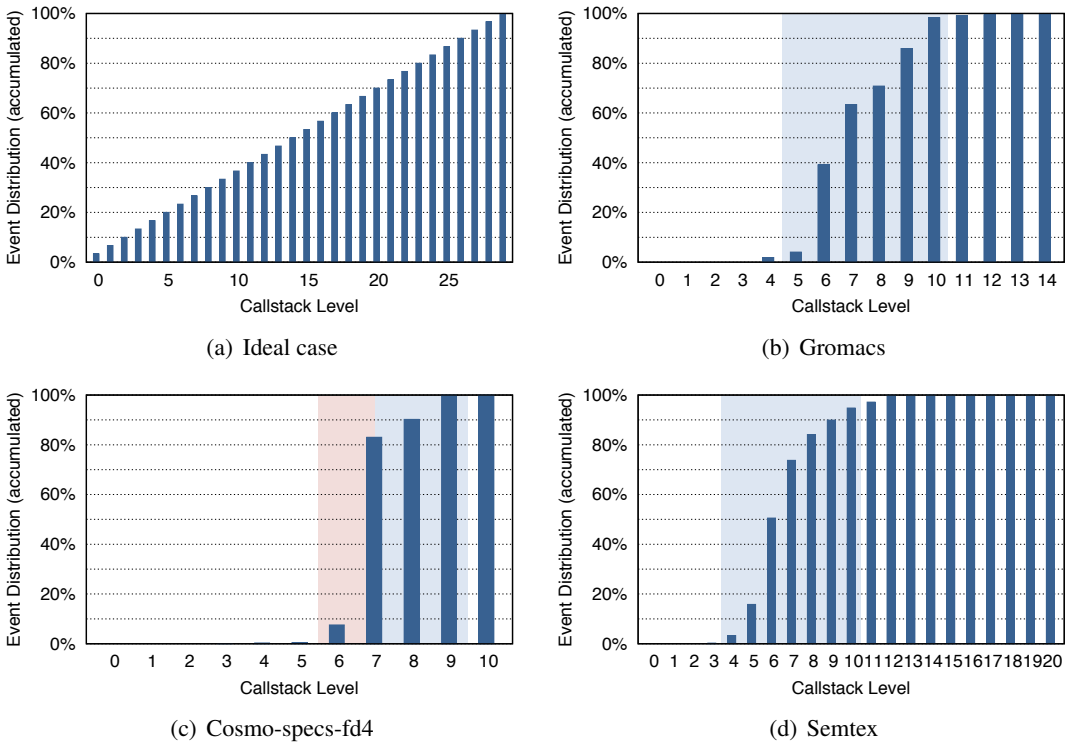
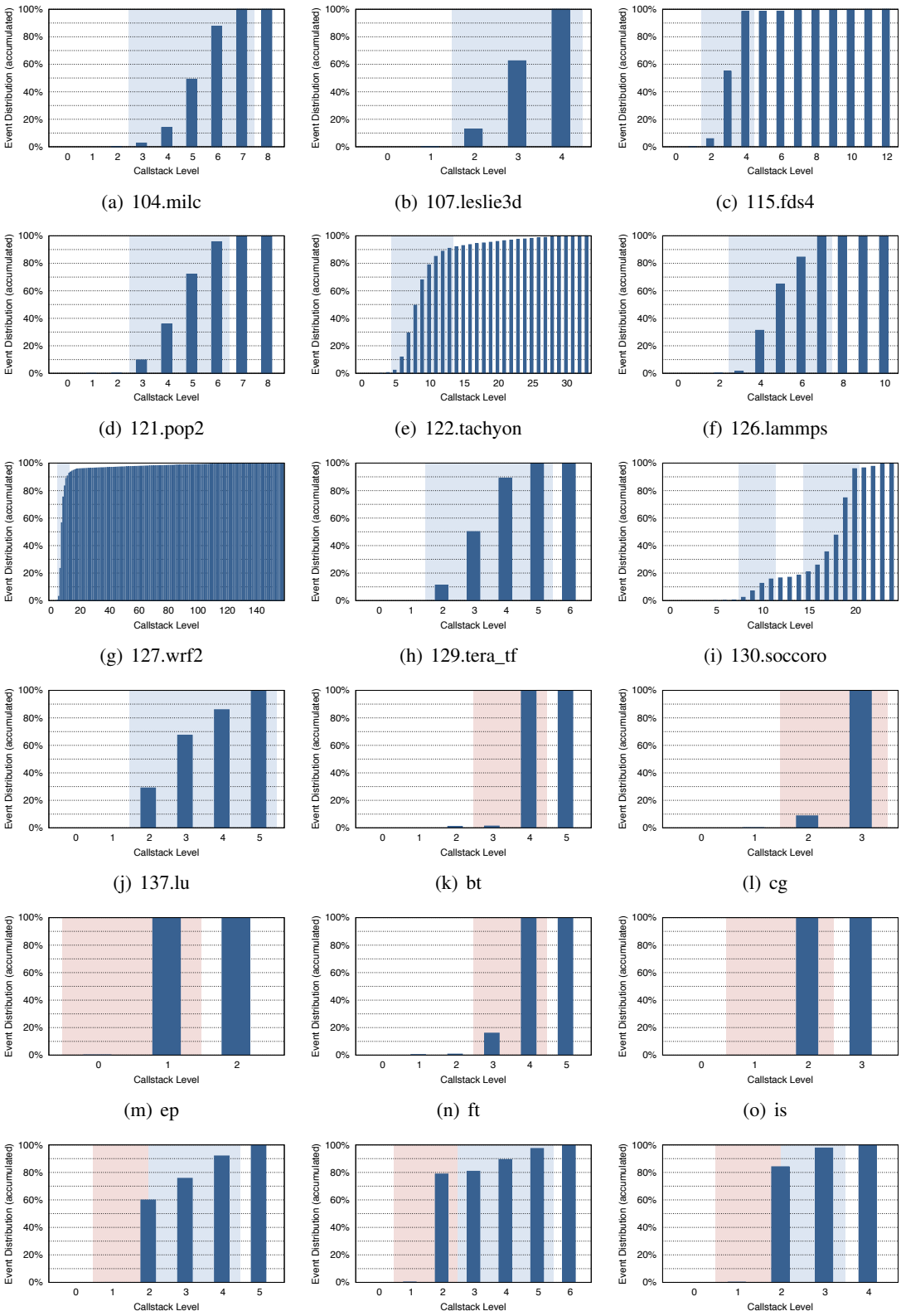


Figure 3.16: Callstack distribution for selected applications.

Figures 3.16 and 3.17 show that for many of the surveyed applications the distribution of events, according to their call stack level, fits well for a reduction by calling depth (marked by a blue background). However, Cosmo-specs-fd4 and most of the NAS Parallel Benchmarks have a very sharp drop in their event distribution (marked by a red background). In this case, a single reduction step discards almost all events recorded so far.



(a) 104.milc (b) 107.leslie3d (c) 115.fds4

(d) 121.pop2 (e) 122.tachyon (f) 126.lammps

(g) 127.wrf2 (h) 129.tera_tf (i) 130.soccoro

(j) 137.lu (k) bt (l) cg

(m) ep (n) ft (o) is

(p) lu (q) mg (r) sp

Figure 3.17: Callstack distribution for SPEC MPI 2007 and NAS Parallel Benchmarks applications.

3.4.4 Reduction by Duration

The fourth and last strategy uses the duration of code regions as criterion for event reduction. While the previous event reduction strategy based on calling depth can deliver promising results for some of the reviewed applications and kernels, for others the lack in granularity leads to too large reduction steps and, therefore, a too large information loss in each step. Therefore, it is essential to further distinguish the events for event reduction. More precisely, it is important to identify those events that contribute less to the overall application behavior.

Having in mind that enter/leave events are the most dominant event class next to performance metrics if they are recorded, the class of enter and leave events provides a good starting point for a further, more detailed reduction strategy. For instance, recording every function call with the same detail is prone to fail, especially, when tiny and often-used functions are monitored, e.g., small helper functions or get and set class methods. An event reduction by duration addresses this impact of high-frequency function calls and presents a method to minimize the amount of high-frequency function calls while still keeping outliers that have an impact on an application's behavior.

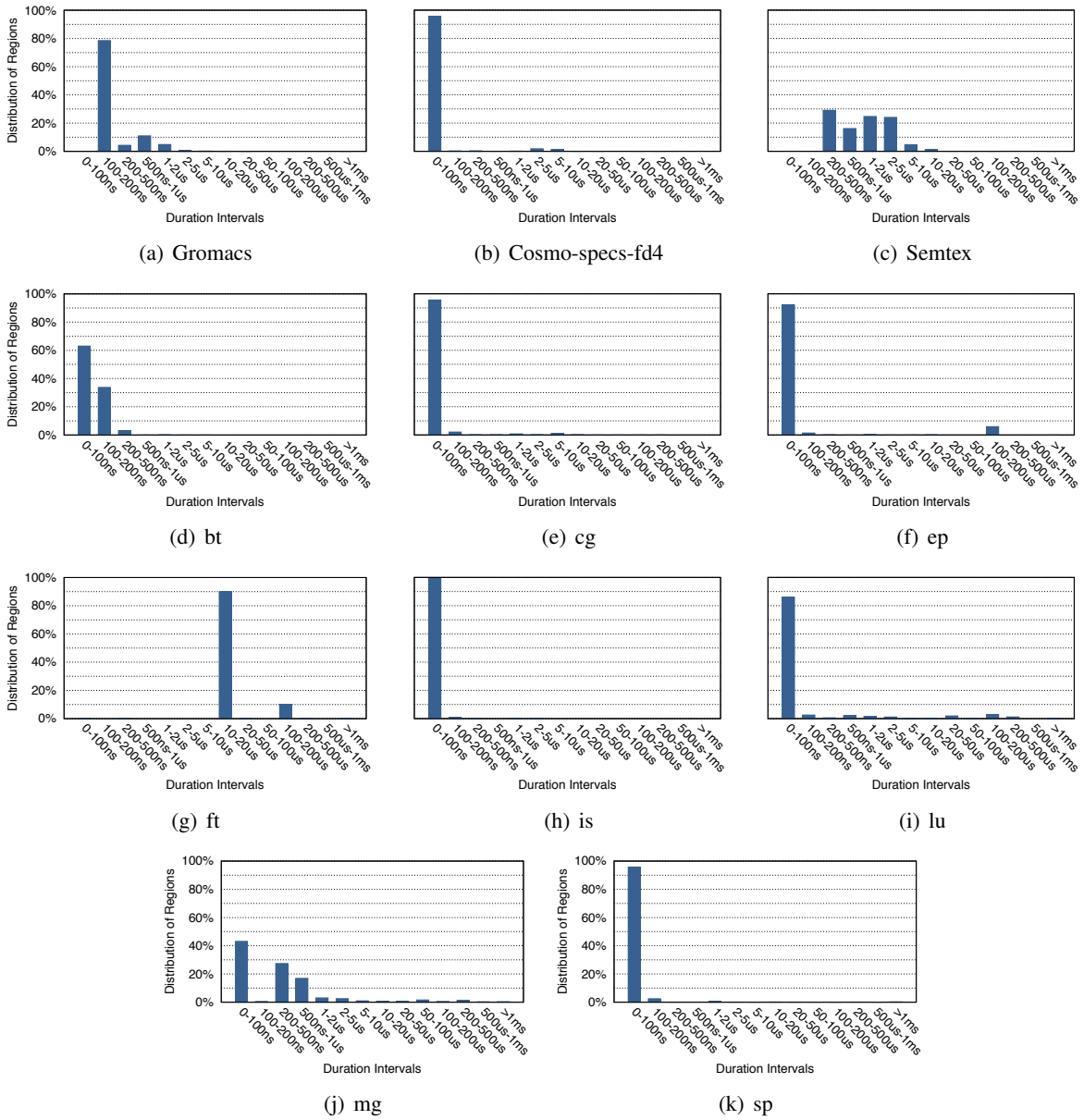
Event-based monitoring records and stores runtime events to provide a detailed and profound analysis. Whereas the following ways are the most prominent to define these events:

- *Compiler instrumentation* inserts compiler-dependent code snippets at the beginning and ending of each code region that provide a monitoring tool information about the current code region,
- *Source-to-source instrumentation* transforms the original application and inserts code snippets at points/regions of interest, e.g., for OpenMP regions and loops,
- *Library instrumentation* intercepts public functions of an external shared library by using a dlopen interception mechanism,
- *Binary instrumentation* modifies the executable either at runtime or before program execution to insert code snippets at function entries and exits, and
- *Manual instrumentation*.

Automated instrumentation techniques like compiler instrumentation are most convenient and easy-to-use. Hence, many event-based monitoring tools use such automated techniques as the default to define events [KRM+12, MKJ+07, GWW+10, SM06, BSC14]. One side effect is, that compiler instrumentation prevents the inlining[2] of tiny and short-running functions such as small helper function or get and set class methods. By itself, a suppressed inlining and recording of these functions provides tools an opportunity to record and analyze an application's behavior very detailed. However, if such short-running functions are heavily called they might overwhelm the capacity of the recording memory buffer while at the same time contribute very little to the overall application behavior.

Figure 3.18 represents the results of an statistical survey with the reviewed applications and kernels that evaluates the distribution of function calls depending on their duration. While the definite distribution deviates slightly, depending on the number of locations and the problem size, the majority of applications shows a clear trend. All applications, except *ft* from the NAS Parallel Benchmarks, use short-running function calls at a very high frequency. Next to that, Figure 3.18 reveals a correlation between a short duration of function calls and a high frequency of occurrence.

[2] Inline expansion or inlining is a compiler optimization that replaces a function call site with the body of the callee, which usually results in improved time and space usage at runtime.



(a) Gromacs (b) Cosmo-specs-fd4 (c) Semtex

(d) bt (e) cg (f) ep

(g) ft (h) is (i) lu

(j) mg (k) sp

Figure 3.18: Duration distribution for selected applications and kernels.

From the results of this survey it can be inferred that short-running high-frequency functions can be eliminated by either tracking their number of occurrences or their duration. Some event tracing tools already provide methods to filter functions depending on their number of occurrences, e.g., filter all calls after a function is recorded n times [MKJ+07, KRM+12] (see Section 3.2). This approach is easy to realize and in some cases suffices the requirement to eliminate most of the frequently called functions. Still, this approach has some disadvantages. First and foremost, there is no heuristic to determine a general number n of function calls after which all further calls are filtered. If it is too high, a lot of unimportant function calls are recorded; if it is too low important function calls might get filtered, as well. While high occurrence often coincides with a short duration, even main routines might get called very often, especially, in long-running iterative applications. For instance, a production run of Gromacs loops over up to one million iterations. In such a case, using a small n results in the complete filtering of all function calls after a certain runtime. Second, the first n calls to an unimportant highly frequent

function are still stored. Especially, if there are multiple of these high-frequency functions, this might already exhaust the recording memory buffer. For instance, [WDKN14] reports 1781 functions that are not inlined in a fully instrumented application. While keeping the first n calls to a function might provide the opportunity to identify that such a function is called very frequently, in an un-instrumented application run such a function would be inlined anyway and, therefore, has a different impact on the application behavior. Third and last, with the method of keeping a maximum of n calls to each function, possible outliers of this function after n calls that actually have an impact on the application behavior are not recorded and, thus, cannot be identified.

Identifying function calls depending on their actual duration does not rely on a correlation between short duration and frequent occurrences of a specific function. In fact, this approach identifies function calls whose duration is within a predefined interval. Hence, it effectively determines all short function calls while still keeping outliers that have an impact on the application behavior. Consequently, this technique overcomes all above mentioned disadvantages of existing filters based on the number off occurrences.

Having this in mind, the duration of code regions can be used to further distinguish events for event reduction. This allows a more fine grained event reduction than with a reduction by calling depth. Since code regions with a very short duration are usually leave nodes in the call tree, or in other words, they occur rather deep in the call stack, applying an event reduction by duration prior to a reduction of the calling depth allows to reduce the step size for the reduction by calling depth.

An alternative to using the duration of code regions for event reduction is to use the duration as runtime filter rule. This way, instead of grouping function calls in intervals for reduction, all function calls shorter than a predefined lower bound are filtered during recording. This method would than be categorized to step one, filtering and selection, rather than step three, event reduction. While the duration of code regions can be used for both, event reduction and filtering, this thesis suggests an implementation as runtime filter. This approach follows the logic that short function calls usually contribute very little to the overall program behavior and, in addition, would probably be inlined anyway without compiler instrumentation. The advantage of a filter by duration is that it effectively filters all short-running function calls while keeping the outliers that have an impact on the application behavior.

3.4.5 Requirements for Event Reduction

This section introduces four strategies for runtime event reduction. These techniques are engaged whenever the internal memory buffer is exhausted; usually this is the point where the memory buffer is either flushed to a file or the measurement is aborted. The primary task of the event reduction is to transparently make space available again by reducing the number of events already stored in the memory buffer. A selection of strategies to efficiently select and reduce events in the memory buffer includes a reduction by temporal order, a reduction based on the class of event, a gradual reduction based on the calling context, and a gradual reduction based on the duration of code regions. With these strategies it can be ensured that the collected event trace data can be efficiently reduced whenever the memory buffer is exhausted, thus making new memory space available for further events and avoiding interaction with the file system at any case, which is the key requirement for an in-memory event tracing workflow.

These strategies require an efficient elimination of events that are already stored in the memory buffer. However, currently non of the presented event tracing tools and libraries [MKJ+07, SM06, KBB+06, EWG+12, WM04] supports such an efficient elimination of events. All current approaches use a flat

continuous memory buffer. Although such a flat continuous memory buffer allows the elimination of events that are already stored, such an elimination would introduce an enormous overhead.
All four event reduction strategies first require the identification of those events that are about to be discarded. For this purpose a specific property of each event needs to be evaluated, which is either its time stamp, its event class, or its calling depth. This would oblige scanning the complete memory buffer for these events. After that, all events located for elimination are discarded and the remaining memory sections are marked as free. Since events occur at a high frequency and are typically only a few bytes small, there are plenty of small free sections scattered across the memory buffer. To gain a continuous free memory section at the end, the remaining non-free memory sections must be collapsed to one memory section, resulting in numerous small memory moves. This approach is extremely expensive and cannot be applied for event reduction during runtime. The next chapter presents an alternative to such a traditional flat memory buffer: the Hierarchical Memory Buffer, which supports event reduction operations with minimal overhead.

3.5 Summary

The nature of event tracing is to provide very detailed information by collecting and storing runtime events, such as function entry and exit or sending and receiving a message. Corresponding event records are stored within internal memory buffers. Although these event records themselves are rather small, they are typically recorded at very high rates, which regularly results in huge generated data volumes that overwhelm the memory buffer capabilities.
This chapter presents an approach to keep the event data for an entire measurement run within a single fixed-sized memory buffer to enable an all in-memory event tracing workflow. This approach is based on three major steps. The first step contains methods to either select events for monitoring before the measurement starts or filter events during runtime, in particular, a novel technique to filter functions call by their duration. The second step is an efficient storage of event tracing data within the memory buffer. This requires a compact encoding and low-overhead compression. This step is built on top of existing encoding methods and presents new enhanced techniques to drastically increase memory efficiency.
The third and last step, event reduction, is entirely different. While these first two steps can significantly reduce memory allocation, they fail the critical criterion for an in-memory workflow: they cannot guarantee that the data of an arbitrary measurement fits into a single memory buffer of fixed size. Event reduction is triggered whenever the memory buffer is exhausted and makes memory space available again by reducing the events already stored within the memory buffer while in the same time introducing minimal overhead. A selection of strategies to efficiently select and reduce events in the memory buffer includes a reduction by temporal order, a reduction based on the event class, a reduction by calling depth, and a reduction based on the duration of code regions.
These three steps allow an in-memory event tracing workflow that meets the previosly identified challenges in event tracing: file system limitations in the number of event files, the huge generated data volumes, and the bias caused by intermediate memory buffer flushes (see Section 2.4).
The following chapter introduces the Hierarchical Memory Buffer, a data structure that allows an efficient application of the discussed event reduction techniques. The enhanced encoding techniques, as well as the event reduction methods combined with the Hierarchical Memory Buffer are evaluated in Chapter 5.

4 The Hierarchical Memory Buffer

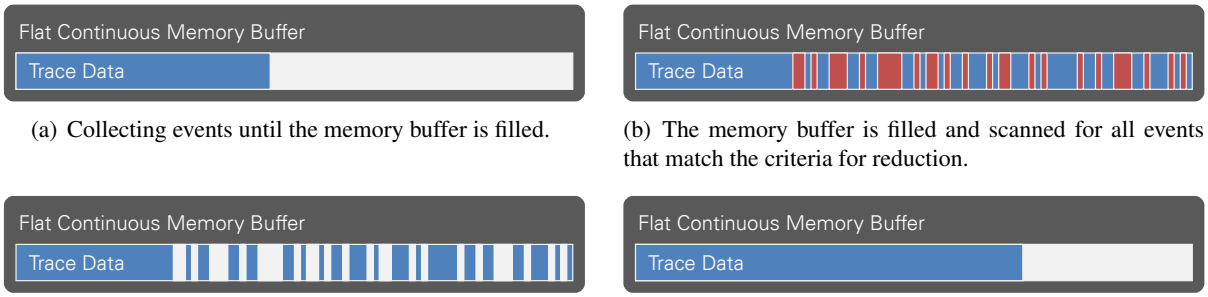
This chapter introduces the Hierarchical Memory Buffer, a data structure that allows to perform event reduction operations with minimal overhead. It examines algorithms for the construction of this data structure, as well as the application of the event reduction strategies and typical analysis techniques. Furthermore, the computational complexity of all algorithms is discussed.

4.1 Memory Event Representation

The introduced event reduction strategies require an efficient identification and elimination of events that are already stored in the memory buffer. However, currently non of the presented event tracing tools and libraries supports such an efficient elimination of events. They all use a flat continuous memory buffer that, although, allowing the elimination of events already stored in the memory buffer, introduces an enormous overhead when engaged. This section is dedicated to more efficient alternatives.

4.1.1 Flat Continuous Event Representation

The majority of event tracing libraries and tools use internal memory representations with a single continuous memory buffer to store recorded events [SM06, MKJ+07, WM04, KBB+06, EWG+12]. Such a memory buffer has a flat layout and stores events in the order of their occurrence, which is equal to a temporal order. A single continuous memory buffer is an efficient data structure to store events during measurement and the order of occurrence represents the natural order to read events from the buffer in a subsequent analysis. However, a flat memory buffer is not capable to represent any hierarchical information other than the order of occurrence. Such additional hierarchy information is, for instance, the call stack or the class of an event to distinguish different metrics or parallel paradigms.
First of all, it is possible to apply the event reduction strategies from Section 3.4 on traditional flat continuous memory buffers but these event reduction operations introduce remarkable overhead. A flat continuous memory buffer stores the recorded events in the order they occurred until the memory buffer is exhausted (see Figure 4.1(a)). When the memory buffer is exhausted the event reduction is triggered. Since all events are scattered over the memory buffer, the entire memory buffer needs to be scanned to find all events that match the criterion for reduction (see Figure 4.1(b)), e.g., all events of the deepest call stack level for a reduction based on the calling depth (see Section 3.4.3). When all events matching the reduction criteria are found, they are discarded and the according memory sections are marked as free (see Figure 4.1(c)). Since events occur at a high frequency and are typically only a few bytes small, there are plenty of small free sections scattered over the whole memory buffer. This leaves a highly fragmented memory buffer that cannot be used for writing further events. Thus, all non-free memory sections need to be moved to collapse the fragmented memory buffer to a single continuous memory segment that leaves a continuous free memory section at the end to store further events (see Figure 4.1(d)).



(a) Collecting events until the memory buffer is filled.

(b) The memory buffer is filled and scanned for all events that match the criteria for reduction.

(c) The memory sections of these events are marked as free.

(d) The remaining non-free memory is collapsed to one memory section to provide a continuous free memory section at the end for new events.

Figure 4.1: Event reduction with a flat continuous memory buffer.

Let n be the number of total events within the memory buffer before the event reduction and let $m \leq n$ be the number of events that match the criterion for reduction. Then the computational complexity of a reduction operation can be expressed as follows. First, the entire memory buffer must be scanned and each event must be evaluated whether or not it matches the reduction criterion, which has a complexity of $\mathcal{O}(n)$. Second, all events that match the reduction criterion are discarded and their reserved memory sections are marked as free, which is in $\mathcal{O}(m)$. Third, all non-free memory sections need be collapsed. Since source and destination of the memory move operation may overlap, this operation requires an additional copy to an intermediate buffer; otherwise the behavior of the memory transfer is undefined [C11]. Nonetheless, the complexity of the memory defragmentation is in $\mathcal{O}(m)$. Therefore, the computational complexity of the entire reduction operation is in $\mathcal{O}(n+m+m) = \mathcal{O}(n)$, with $m \leq n$. Since a memory buffer, depending on its size, can contain several million events such a reduction operation introduces a remarkable overhead when using a traditional flat continuous memory representation.

4.1.2 Flat Partitioned Event Representation

To avoid a costly reduction operation it is necessary to sort all events prior to the event reduction operation. This means, all events need to be sorted by the different reduction criteria, e.g., time of occurrence, calling depth, event class, and code region duration, while they are recorded. A translation to a continuous flat memory buffer requires a partitioning of the memory buffer in multiple partitions; one for each distinct value of the reduction criteria. For simplification the following section considers only one reduction criterion, e.g., the calling depth. Figure 4.2 demonstrates such a partitioning within a flat continuous memory buffer. A partitioned memory buffer contains a header that keeps partitioning data, e.g., pointers to the beginning and the current write position of each partition, and p partitions to store the actual event trace data.

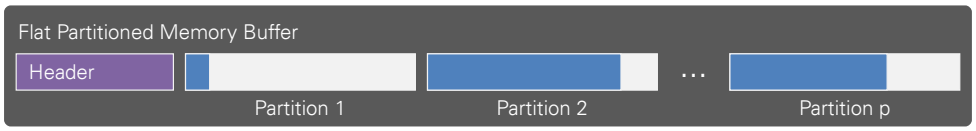


Figure 4.2: Flat partitioned event representation including a header and one partition for each value of the event reduction criterion.

Since all events are presorted into their according partition, the identification of all events that match a certain reduction criteria can be done without the need to read the entire memory buffer, thus, the complexity of the identification is in $\mathcal{O}(1)$. Discarding all events in a certain partition q and marking the memory as free is also in $\mathcal{O}(1)$. However, the freed partition needs to be divided in $p-1$ sections that are distributed to the remaining partitions in order to use the freed memory to store further events in the remaining partitions. Therefore, all partitions need to be moved and the header needs to be updated. Since source and destination of the memory move operation may overlap, this operation requires an additional copy to an intermediate buffer, as well. The complexity of the memory reorganization is in $\mathcal{O}(p)$, with p representing the number of partitions. Therefore, the complexity of the entire reduction operation is in $\mathcal{O}(1+1+p) = \mathcal{O}(p)$. In worst case, e.g., recursive function calls with a recursion depth $r \geq n$ (the maximum number of events in the buffer), the number of partitions p equals the number of events n. In that case, the complexity is again in $\mathcal{O}(n)$.

In addition, this worst case example demonstrates a critical restriction of such a partition memory buffer: when the number of partitions is not limited by an upper bound – which is usually the case – the partition needs to be reorganized whenever an event matching a not already represented value of a reduction criterion needs to be stored. Such a reorganization is the reverse operation to the reorganization within a reduction operation and, thus, also in $\mathcal{O}(p)$.

Another disadvantage of this approach is the potential imbalance in the utilization of the partitions. With the *memory balance* $\mathcal{M}$ between different partitions is defined as:

$$\text{Memory balance } \mathcal{M} = \frac{\text{Average memory utilization}}{\text{Maximum memory utilization}} \tag{4.1}$$

The memory utilization is a function $U : \mathbb{N} \to [0,1] \in \mathbb{R}$ that maps each partition to the fraction of bytes used divided by the total number of bytes. Thus, the memory balance for a fixed number of partitions p can be expressed as:

$$\mathcal{M} = \frac{\frac{1}{p}\sum\limits_{i=1}^{p} U(i)}{\max\limits_{1\leq i\leq p}\{U(i)\}}$$

The memory balance is especially important when one partition is exhausted and, therefore, a reduction operation is triggered. In worst case, one partition is exhausted while all other partitions are still empty:

$$\mathcal{M} = \frac{\frac{1}{p}\sum\limits_{i=1}^{p} U(i)}{\max\limits_{1\leq i\leq p}\{U(i)\}} = \frac{\frac{1}{p}\Big((p-1)\cdot 0 + 1\cdot 1\Big)}{1} = \frac{1}{p}$$

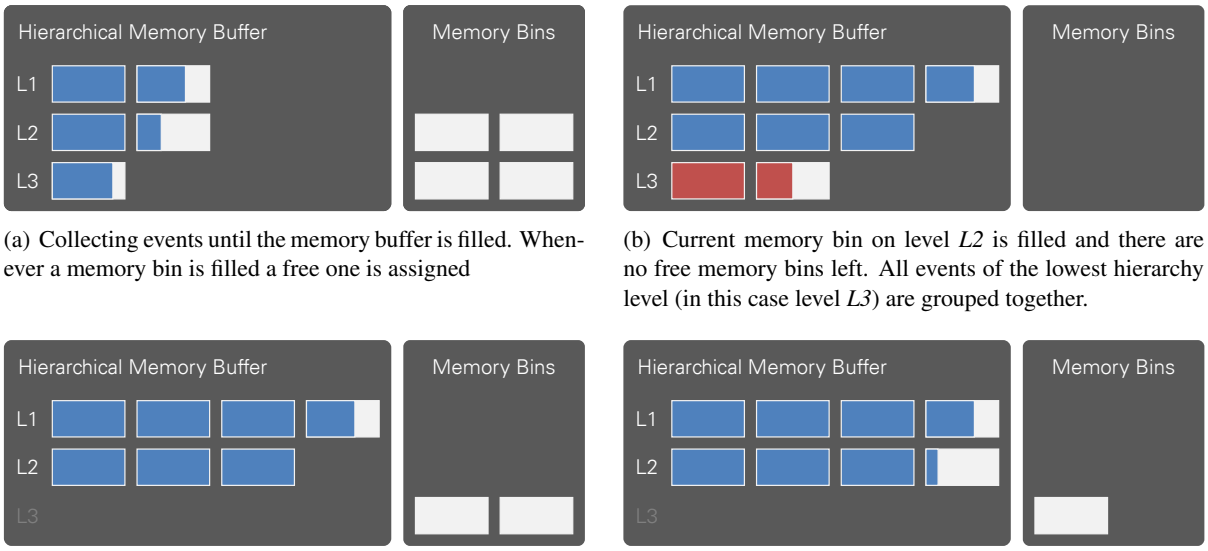
In worst case the memory balance and, therefore, the memory efficiency is $\frac{1}{p}$, i.e., a reduction operation is triggered although there is only $\frac{1}{p}$th of the total memory used.

A partitioned memory representation reduces the complexity for a single event reduction operation from $\mathcal{O}(n)$ to $\mathcal{O}(p)$. However, the inflexible fixed partitioning of the buffer imposes additional overhead and provides a lower memory efficiency, which leads to premature event reduction operations.

4.1.3 Hierarchical Event Representation

In contrast to the previous static flat memory representation this section introduces a hierarchical memory representation. The hierarchical memory representation is organized as a multi-dimensional array, where each *hierarchy dimension* represents one possible hierarchical order with a flexible number of different values within that hierarchical order, called *hierarchy levels*. In the context of event reduction, for instance, one dimension can represent the calling depth and another the event class.

Instead of one huge memory chunk, the total memory allocation for the according memory buffer is divided in plenty of small memory sections, called *memory bins*. These memory bins can be dynamically distributed to any hierarchy level in each dimension. Whenever an event needs to be stored at a certain hierarchy level and there is either no memory bin assigned or the current memory bin is exhausted, a free memory bin is distributed to this hierarchy level; assuming there are further free memory bins left.



(a) Collecting events until the memory buffer is filled. Whenever a memory bin is filled a free one is assigned

(b) Current memory bin on level *L2* is filled and there are no free memory bins left. All events of the lowest hierarchy level (in this case level *L3*) are grouped together.

(c) All memory bins assigned to level *L3* are revoked and all events are automatically discarded.

(d) One of the free memory bins is assigned to level *L2* to store further events.

Figure 4.3: Event reduction with a hierarchical event representation.

Figure 4.3 demonstrates the event reduction with such a hierarchical event representation. Again, for simplification this example considers at first only one reduction criterion, e.g., the calling depth. Thus, the according memory buffer's layout is an one-dimensional array. When the first event needs to be stored, usually on call stack level $L1$, no memory bin has been assigned to this hierarchy level, so far. Thus, the memory buffer checks if there is a free memory bin available, which is true in this case, and one memory bin is assigned to the hierarchy level $L1$, so, the event can be stored. If an event needs to be stored on a different hierarchy level, a free memory bin is assigned the same way. The same applies, when on any hierarchy level the current memory bin is exhausted. After some time, this leads to a situation like in Figure 4.3(a): Five memory bins are assigned to the hierarchy levels $L1$ - $L3$ and four free memory bins are available. Hence, four additional memory bins can be assigned to the hierarchy levels. After that, all memory bins are assigned and there are no free memory bins available anymore. This leads to the situation in Figure 4.3(b): An event needs to be stored at the hierarchy level $L2$ but there are no free memory bins available. At this point, the event reduction is triggered and all events of a certain hierarchy

level are discarded, for instance, all events of the deepest call stack level, in this case, level $L3$. Since all events are already sorted by their call stack level the event reduction operation can be done with minimal costs. The event reduction just revokes all memory bins assigned to the hierarchy level $L3$ and adds them again to the pool of free memory bins. In addition, the hierarchy level $L3$ is marked as closed, so, all future events on this hierarchy level are discarded right away. After that, the two revoked memory bins are available again (see Figure 4.3(c)). Therefore, one of them can be assigned to the hierarchy level $L2$ and the event that triggered the event reduction can be stored (see Figure 4.3(d)).

Next to the layout as one-dimensional array as for the example above, the hierarchical event representation can be organized as a multi-dimensional array, as well. In that case, the event reduction can be applied on a complete row or column within the multi-dimensional array. This way, a hierarchical memory representation is able to support all event reduction techniques simultaneously.

Since all events are presorted into their according partitions, the complexity for the identification of all events that match a certain reduction criterion is like for the partitioned memory representation in $\mathcal{O}(1)$. The complexity to discard all events on a certain hierarchy level and marking the memory bins as free depends on the number of memory bins b of that level and can be realized in $\mathcal{O}(b)$. Revoking all memory bins and adding them again to the pool of available memory bins is in $\mathcal{O}(b)$, as well. Therefore, the computational complexity of the entire reduction operation is also in $\mathcal{O}(b)$.

In contrast to the partitioned memory representation such a hierarchical event representation is capable to contain an unlimited number of hierarchy levels. Though, for a practical implementation the number of hierarchy levels is limited by the number of available memory bins, which depends on the size of total memory allocation, the size of the memory bins, and the number of hierarchy dimensions:

$$\text{Maximum hierarchy level } = \frac{\text{Total memory allocation size}}{\text{Size of memory bins} \times \text{Hierarchy dimensions}}$$

Since the total memory allocation is partitioned into fixed-sized memory bins this approach is also prone to memory imbalances like the partitioned memory representation. However, because there can be multiple memory bins assigned to each hierarchy level, instead of a single partition, the memory imbalance can be drastically reduced.

For a fixed number of memory bins b and hierarchy partitions p = hierarchy levels $\times$ dimensions, with $b \geq p$ the memory balance defined in Equation 4.1 can be adapted to the number of memory bins:

$$\mathcal{M} = \frac{\frac{1}{b}\sum\limits_{i=1}^{b} U(i)}{\max\limits_{1 \leq i \leq b}\{U(i)\}}$$

Again, considering the memory balance at the point where one partition is exhausted and a reduction operation is triggered, in worst case, i.e., all other partitions are still empty[1], the memory balance is:

$$\mathcal{M} = \frac{\frac{1}{b}\sum\limits_{i=1}^{b} U(i)}{\max\limits_{1 \leq i \leq b}\{U(i)\}} = \frac{\frac{1}{b}\Big((p-1)\cdot 0 + (b-p+1)\cdot 1\Big)}{1} \geq \frac{b-p}{b} = 1 - \frac{p}{b} \tag{4.2}$$

[1]Since the memory bins are only distributed if an event actually needs to be stored in a hierarchy partition, technically, empty memory bins do not exist, i.e., at least one event is stored within each memory bin. Therefore, the actual memory efficiency is slightly higher depending on the ratio of event size to memory bin size.

Hence, the memory balance and, therefore, the memory efficiency depends on the number of memory bins. For instance, if there are ten times more memory bins than hierarchy partitions, the memory balance is at least 0.9 and, if there are 100 times more memory bins, the memory balance is at least 0.99, i.e., at the point where a reduction operation is triggered at least 90 % or 99 %, respectively, of the total memory is utilized, which provides a major benefit over a fixed partitioned event representation.

Consequently, the memory efficiency of such a hierarchical memory representation mainly depends on the number of available memory bins in proportion to the number of partitions. Since, the total memory allocation is fixed, the number of memory bins results from the size of the memory bins with smaller memory bins leading to a higher memory efficiency. Therefore, the size of the individual memory bins is a crucial parameter of a hierarchical event representation and is evaluated in detail in Section 5.3.1.

4.2 The Hierarchical Memory Buffer Data Structure

A hierarchical memory representation as discussed above requires a data structure that allows access to multiple hierarchy dimensions and an arbitrary number of hierarchy levels within each dimension. In addition, for each hierarchy level the memory is distributed in the form of an arbitrary number of small memory bins to optimize memory efficiency. A data structure that supports such a hierarchical event representation must consider the typical access modes to store and read data, in particular, for the storage of event tracing data with respect to the above discussed event reduction operations.

The event reduction operations discussed in Section 3.4 require a two-dimensional memory buffer layout: one dimension representing the different event classes, e.g., code regions, communication, and the other dimension representing the calling depth of each event. As mentioned before, the method to distinguish events by the duration of the according code region provides another strategy for event reduction. In this case, a third dimension representing different intervals of code region duration is required. However, as stated in Section 3.4.4 this thesis suggests to use the duration of a code region as runtime filter because short function calls generally contribute very little to the overall program behavior and, in addition, would probably be inlined anyway without compiler instrumentation.

Since events from different event classes may occur at any given point within an application the data structure for the first dimension requires efficient random access. The same applies for events in terms of their call stack stack level. While events of code regions (enter, leave) always access the neighboring call stack level, other events occur on arbitrary levels within their event class. Thus, the second dimension representing the call stack levels requires efficient random access, as well.

To allow a hierarchical event representation with random access to each hierarchy dimension and each hierarchy level, this section introduces a novel data structure called *Hierarchical Memory Buffer*. The Hierarchical Memory Buffers uses sequence containers like arrays that can change in size, also known as *vectors*, to represent each dimension. Vectors, like arrays, use continuous storage locations for their entries, which provides random access with constant time complexity. In contrast to arrays, vectors can change their size dynamically to handle an arbitrary number of elements. To grow in size the underlying array needs to be reallocated and the elements moved, which is an expensive task in terms of processing time. Hence, vectors in the Hierarchical Memory Buffer do not reallocate each time a new entry is added but allocate additional memory for multiple entries in advance. Since there are only a few different event classes and Figures 3.16 and 3.17 provide heuristics for the number of call stack levels, a reallocation can

be avoided entirely for most applications. This way, all elements of the different hierarchy dimensions and hierarchy levels, called *hierarchy entries*, can be accessed with constant time complexity in the Hierarchical Memory Buffer.
Accessing the event data storage within each hierarchy entry in the form of memory bins is strictly sequential for writing, as well as reading. Furthermore, the number of memory bins must be highly dynamic to achieve an optimal memory balance. Therefore, the memory bins are arranged as single-linked lists. Each hierarchy entry manages the list of memory bins and keeps direct access to the current positions for writing and reading within the current memory bin. Figure 4.4 illustrates the Hierarchical Memory Buffer data structure in a two-dimensional layout.

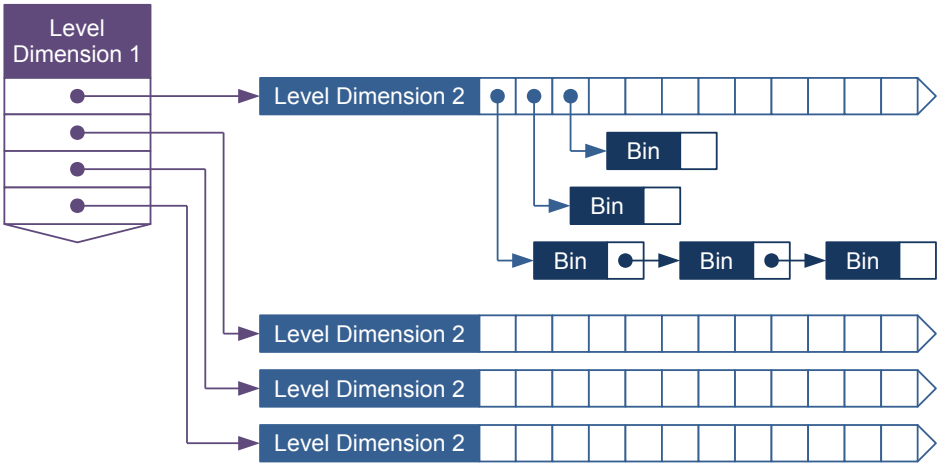


Figure 4.4: Illustration of a two-dimensional Hierarchical Memory Buffer data structure. Each dimension is realized as a vector sequence container while the memory bins for each hierarchy level is arranged as single-linked list.

A such composed data structure allows sequential write and read access with constant time complexity while allowing an unlimited number of hierarchy dimensions and hierarchy levels as well as a highly dynamic distribution of memory bins for optimized memory utilization.

4.3 Construction of the Hierarchical Memory Buffer

The Hierarchical Memory Buffer is designed to represent runtime events of a single location. For parallel applications multiple Hierarchical Memory Buffers are used; one for each location. The following section focusses on the construction of a single Hierarchical Memory Buffer. Since there are no correlations between Hierarchical Memory Buffers of different locations, the construction as well as write/read access can be performed entirely in parallel without any dependencies.
The construction of the Hierarchical Memory Buffer relies on two properties for all events:

1. *Uniqueness*: All hierarchy levels, e.g., event classes and call stack levels, are disjunct sets of events, thus, each event can be distinctively assigned to exactly one hierarchy level.

2. *Correct Nesting*: All events that occur in pairs are properly nested, i.e., for each two pairs of events (a, b) and (c, d) the order of the events is either $a < c < d < b$ or $c < a < b < d$. This property is also known as the stack property.

Furthermore, the construction of the Hierarchical Memory Buffer distinguishes between three types of events: region *enter* events, region *leave* events, and *non-region* events. At the start of the construction the Hierarchical Memory Buffer is empty, i.e., no memory bins are assigned, and the call stack pointer equals zero. A write operation for every event alters the Hierarchical Memory Buffer, as described, by the algorithms in Figure 4.5. Whereas the *assignMemory* function can be expressed by the algorithm in Figure 4.6. Thereby, the two above properties ensure a correct construction of the Hierarchical Memory Buffer. The uniqueness property allows a clear differentiation in region enter, region leave, and non-region events. The correct nesting property guarantees that all events within one code region including its enter and leave events are on the same hierarchy level within their according hierarchy dimension.

Function writeRegionEnterEvent

```
1 increment call stack pointer
2 assignMemory( event class, call stack pointer )
3 write event to current memory bin
4 update write pointer
```

Function writeRegionLeaveEvent

```
1 assignMemory( event class, call stack pointer )
2 write event to current memory bin
3 update write pointer
4 decrement call stack pointer
```

Function writeNonRegionEvent

```
1 assignMemory( event class, call stack pointer )
2 write event to current memory bin
3 update write pointer
```

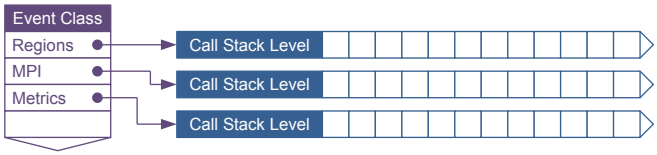
Figure 4.5: Algorithms to alter the Hierarchical Memory Buffer for region enter, region leave, and non-region events.

Function assignMemory(event class, call stack level)

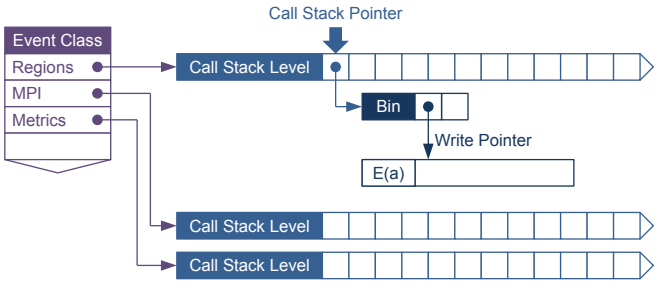
```
1 if position (event class, call stack level) has no memory bin then
2 |   request new memory bin
3 |   set new memory bin as current memory bin and update write pointer
4 else
5 |   if current memory is filled then
6 |   |   request new memory bin
7 |   |   set new memory bin as current memory bin and update write pointer
8 |   end
9 end
```

Figure 4.6: Algorithm to assign memory bins.

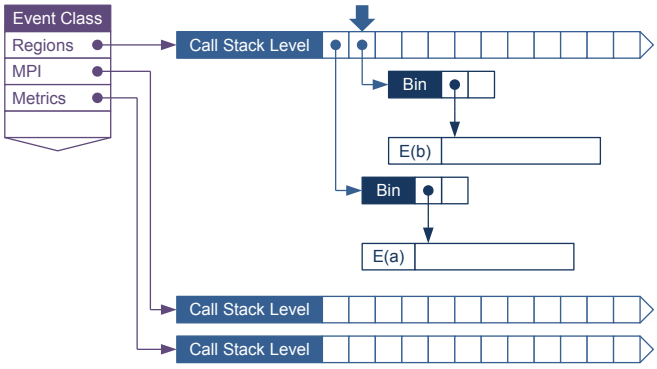
Figure 4.7 demonstrates the construction process for the following event sequence: enter region a, enter region b, leave region b, enter region b, leave region b, enter region b, send message, leave region b, and leave region a.



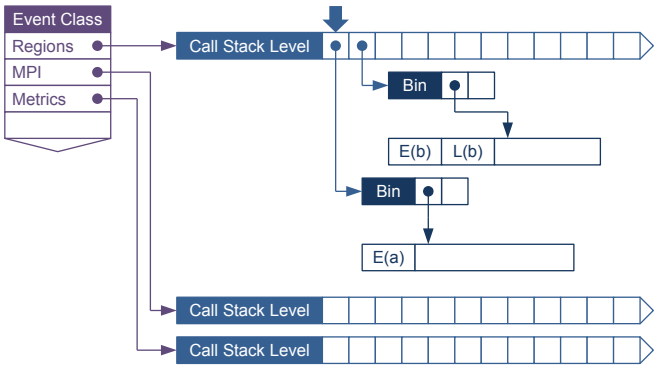
(a) At the start the data structure is a two-dimensional empty array with event classes in the first dimension and call stack levels in the second dimension.



(b) Region a is entered. The call stack pointer is incremented to the first level. At position (Regions,1) there is no memory bin available, thus, one memory bin is distributed, the enter record is written, and the write pointer is updated.

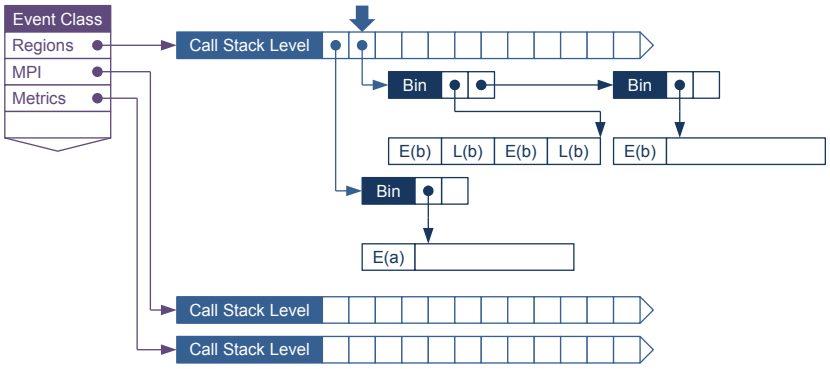


(c) Region b is entered. The call stack pointer is incremented to the second level. At position (Regions,2) there is no memory bin available, thus, one memory bin is distributed, the enter record is written, and the write pointer is updated.

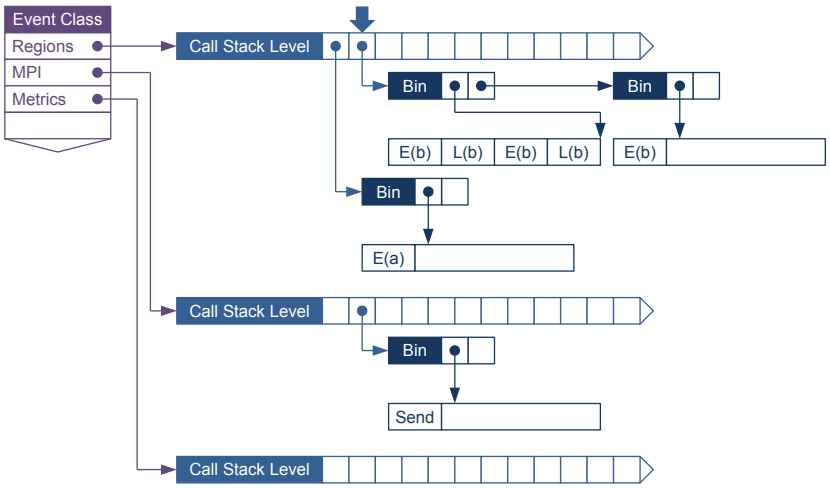


(d) Region b is left. At position (Regions,2) there is a memory bin available, thus, the leave record is written and the write pointer is updated. The call stack pointer is decremented to the first level.

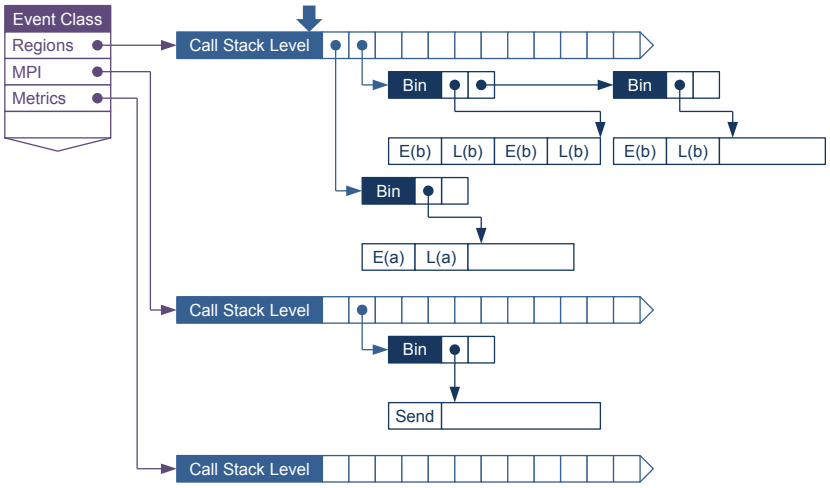
Figure 4.7: Construction of the Hierarchical Memory Buffer (Part I).



(e) Region b is entered. The call stack pointer is incremented to the second level. At position (Regions,2) there current memory bin is filled, thus, the next memory bin is distributed, the enter record is written, and the write pointer is updated.



(f) A message is sent. At position (MPI,2) there is no memory bin available, thus, one memory bin is distributed, the send record is written, and the write pointer is updated.



(g) After leaving region b and a the call stack pointer is at its initial position.

Figure 4.7: Construction of the Hierarchical Memory Buffer (Part II).

4.4 Reduction Techniques with the Hierarchical Memory Buffer

The main purpose of the Hierarchical Memory Buffer is to enable the event reduction operations presented in Section 3.4 in an efficient way. This section discusses the different event reduction operations on the Hierarchical Memory Buffer data structure. Without loss of generality, this section focuses on a two-dimension layout of the hierarchical buffer with the different event classes in the first dimension and the call stack levels in the second dimension (see Figure 4.7(a)). For the sake of comprehensibility, the levels of the first dimension – the different event classes – are referred to as *rows* and the levels of the second dimension – the call stack levels – are referred to as *columns*.

4.4.1 Reduction by Order of Occurrence

A reduction by the order of occurrence reduces events by the order of their occurrence. Basically, this includes three different reduction strategies: a) Keep events from a starting point, b) keep events until an end point , and c) keep the first or the last events within a specific application phase. Whereas a) and b) are a special case of c) in which the specific application phase equals the entire application runtime (see Section 3.4.1). Keeping the leading events can be easily realized by stopping the storage of all further events once the memory buffer is exhausted. Keeping the trailing events requires a cyclic buffer layout that starts overwriting events in the front of the buffer whenever the end of the buffer is reached. For all formats that do not use a fixed size for all events, the overwriting can only be done for complete segments of the buffer. In this case, the memory buffer is divided in s segments where each segment has a defined starting point for the first event. When the memory buffer is exhausted, the first segment is cleared and the next event is stored in the first segment at the starting position. When the first segment is filled, the second segment is cleared and so on.
For the Hierarchical Memory Buffer each segment is represented by its own Hierarchical Memory Buffer (Figure 4.8). Thus, when all the sub-buffers are exhausted, the first memory buffer is cleared, and the next event is stored in this first sub-buffer and so on. All three strategies do not require the Hierarchical Memory Buffer because for typical memory buffers the events are already ordered by their occurrence. Hence, all three strategies can be realized similar to the Hierarchical Memory Buffer.

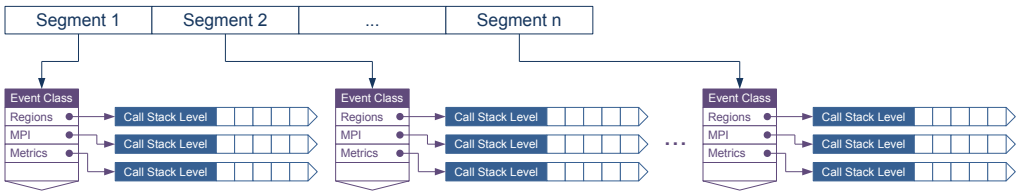


Figure 4.8: Cyclic segmented buffer layout with Hierarchical Memory Sub-buffers.

The complexity of keeping leading events is $\mathcal{O}(1)$ since the storing of events can be just stopped. Keeping trailing events requires to reset one sub-buffer whenever the previous sub-buffer is exhausted – starting when all sub-buffers are filled for the first time. For such a reset of a sub-buffer all distributed memory bins to all event classes and all call stack levels of this sub-buffer need to be revoked. This depends on the function *revokeMemoryBins* (Figure 4.9) that iterates over all memory bins of a given event class and call stack level and deallocates each memory bin. Let b_s be the number of memory bins within the sub-buffer s. Then the complexity of the reduction operation is $\mathcal{O}(b_s)$.

```
Function revokeMemoryBins( eventClass, callStackLevel )
    Data: Event class and call stack level where the memory bins are revoked.
    Result: Hierarchical Memory Buffer where all memory bins are revoked for the given event
            class and call stack level.
1 current = first memory bin for eventClass and callStackLevel
2 while current == NULL do
3     next = current.next
4     free current
5     current = next
6 end
```

Figure 4.9: Algorithm to revoke all memory bins for a given event class and call stack level.

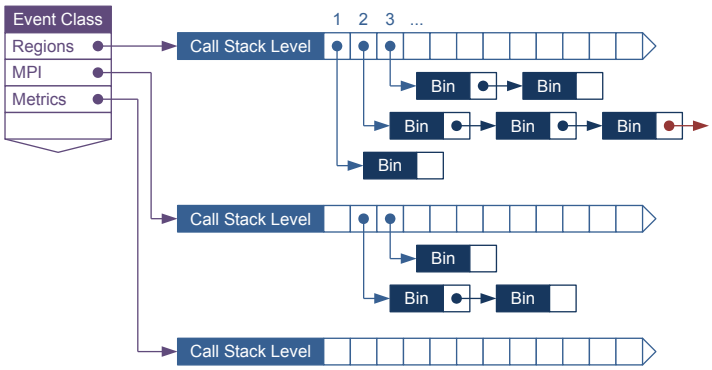
4.4.2 Reduction by Event Class

The reduction by event class reduces all events of a single event class, e.g., all performance metrics or all communication events (see Section 3.4.2). Since all events are presorted based on their event class, the reduction operation equals a reduction of one row in the Hierarchical Memory Buffer. Figure 4.10 demonstrates the basic algorithm for the reduction of a single event class. It depends on the function *revokeMemoryBins* that deallocates all memory bins of a given event class and call stack level. After revoking all memory bins the event class is marked as closed, thus, no further events are stored for that event class. Figure 4.11 depicts this event class reduction operation for a simple scenario.

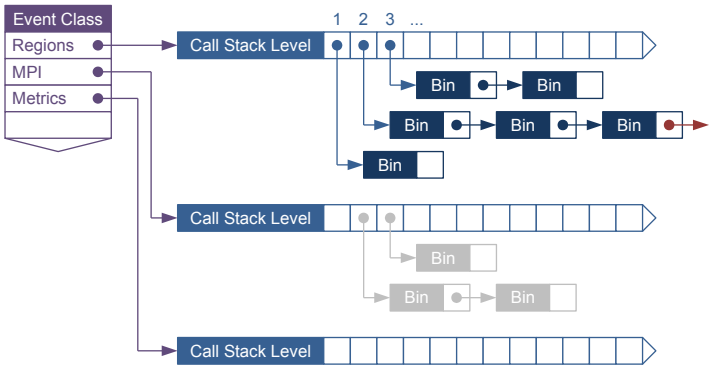
```
Function reduceEventClass( eventClass )
    Data: Event class to be reduced.
    Result: Hierarchical Memory Buffer without the row eventClass.
1 foreach call stack level do
2     revokeMemoryBins( eventClass, call stack level )
3 end
4 Mark eventClass as closed
```

Figure 4.10: Algorithm to reduce an complete event class.

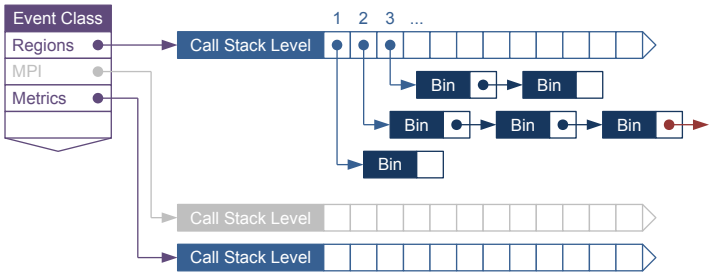
Hence, the complexity of the reduction operation is dependent on the number of call stack levels and the number of memory bins per call stack level. Let $b_{c,l}$ be the number of memory bins assigned to the element with the event class c and the call stack level l. Let further $b_c = \sum_{l=1}^{max} b_{c,l}$ be the number of memory bins assigned to the event class c. Then the complexity of the reduction operation is $\mathcal{O}(b_c)$.



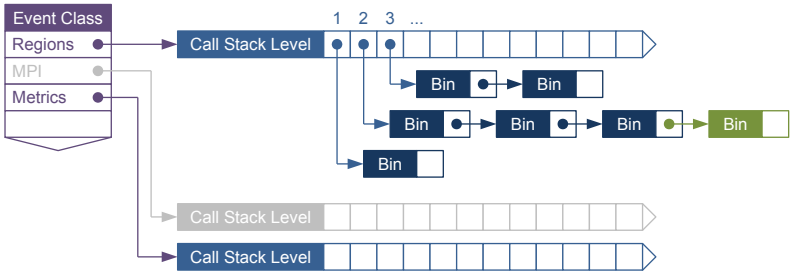
(a) Starting point for the reduction operation is a request for a new memory bin for region events on call stack level two (red arrow) that cannot be satisfied because there are no free memory bins available.



(b) All memory bins distributed to the event class "MPI" have been revoked.



(c) The event class "MPI" is marked as closed. All future communication events will not be stored.



(d) After the reduction operation the revoked memory bins are available for distribution again. One of them is distributed to the region event class on call stack level two to satisfy the initial write request.

Figure 4.11: Reduction by event class on the Hierarchical Memory Buffer.

4.4.3 Reduction by Calling Depth

The event reduction by calling depth reduces all events of the deepest call stack level (see Section 3.4.3). Since all events are presorted based on their call stack level, the reduction operation equals a reduction of one column in the Hierarchical Memory Buffer. Figure 4.12 demonstrates the basic algorithm for the reduction of the deepest call stack level.

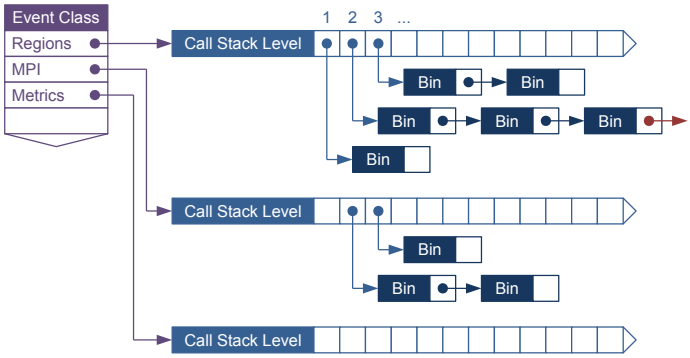
Function reduceHighestCallStackLevel()

Result: Hierarchical Memory Buffer without the column of the highest call stack level.

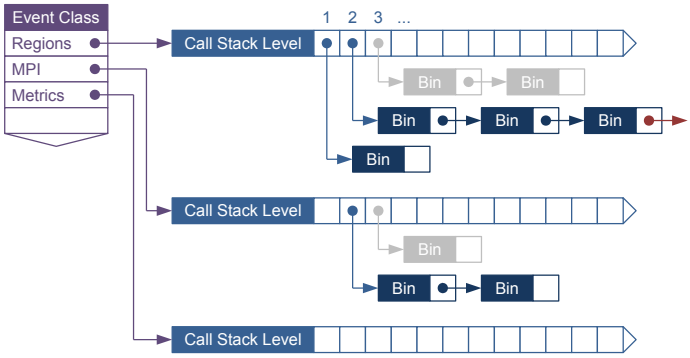
```
1 callStackLevel = deepest non-empty call stack level
2 foreach event class do
3 |   revokeMemoryBins( event class, callStackLevel )
4 end
5 Mark all call stack level ≥ callStackLevel as closed
```

Figure 4.12: Algorithm to reduce deepest call stack level.

Again, the event reduction relies on the *revokeMemoryBins* function to deallocate all assigned memory bins of the deepest call stack level. At the end, all call stack levels deeper than or equal to the currently deepest call stack level are marked as closed, i.e., no further events are stored for this or a deeper call stack level. Figure 4.13 depicts this event reduction operation for another scenario.

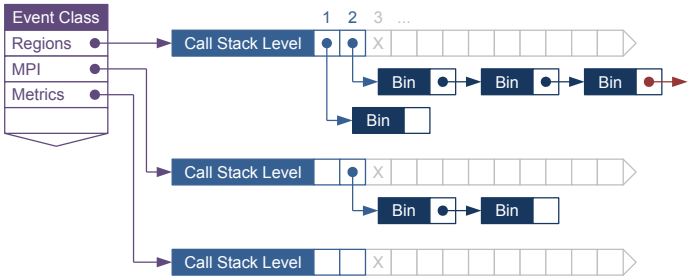


(a) Starting point for the reduction operation is a request for a new memory bin for region events on call stack level two (red arrow) that cannot be satisfied because there are no free memory bins available.

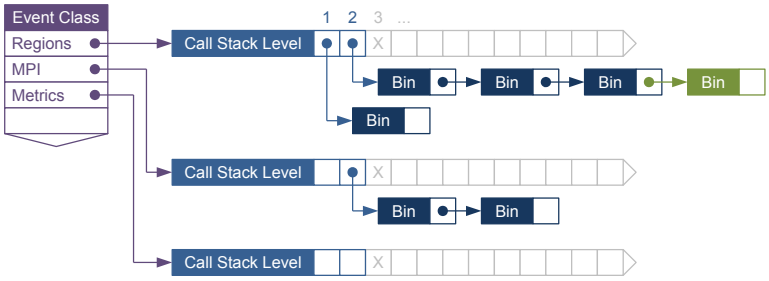


(b) All memory bins distributed to the highest call stack level have been revoked.

Figure 4.13: Reduction by calling depth on the Hierarchical Memory Buffer (Part I).



(c) All call stack level greater than or equal to three are marked as closed. All future events on these call stack level will not be stored.



(d) After the reduction operation the revoked memory bins are available for distribution again. One of them is distributed to the position of the initial write request.

Figure 4.13: Reduction by calling depth on the Hierarchical Memory Buffer (Part II).

Marking all call stack levels greater than or equal to the current highest call stack level as closed can be realized independently from the number of call stack levels, by setting the value for a low pass filter for the call stack levels to the currently deepest call stack level minus one. Therefore, the reduction operation is again only dependent on the complexity of the *revokeMemoryBins* function, which relies on the number of event classes and the number of memory bins per event class. Again, let $b_{c,l}$ be the number of memory bins assigned to the element with the event class c and the call stack level l. Let further $b_l = \sum_{c=1}^{max} b_{c,l}$ be the number of memory bins assigned to the call stack level l. Then the complexity of the reduction operation is $\mathcal{O}(b_l)$.

4.4.4 Reduction by Duration

An event reduction by the duration distinguishes events by the duration of the according code region. In this case, a third dimension representing different intervals of code region durations is required. An algorithm for an event reduction of duration intervals can be defined similar to the previous algorithms. Instead of one column or row, a reduction operation would reduce a horizontal or vertical plane in a three-dimensional cube. All previous methods must be extended to three dimensions accordingly, which does not change the computational complexity because the event reduction operation only depends on the number of revoked memory bins. The number of revoked memory bins, however, increases with the number of different duration intervals.

At any rate, as stated in Section 3.4.4, this thesis suggests to use the duration of a code region as runtime filter because short function calls generally contribute very little to the overall program behavior and, in addition, are inlined anyway without compiler instrumentation. For such a runtime filter, the start and end time of each code region is evaluated and depending on the duration the code region is either

stored or discarded. Within the Hierarchical Memory Buffer all enter-leave pairs are grouped together, i.e., each enter event is directly followed by the according leave event. To determine the duration of a function call such an approach only requires to store the time and position before the last enter event of each call stack level. In the code region leave event's write routine the duration can be evaluated by comparing the time stamp of the leave event with the stored time stamp of the enter event. If a code region's duration is too short, the write position is simply reset to the stored position before the last enter event. Otherwise, the leave event is written as before. Figure 4.14 shows the modifications to the write enter and write leave operations presented in Figure 4.5. The algorithm for writing non-region events needs no alterations. This approach effectively filters all short-running functions while keeping the outliers that have an impact on the application behavior.

Function writeRegionEnterEvent

```
1 Increment call stack pointer
2 AssignMemory( event class, call stack pointer )
3 Store current write position and time of enter
4 Write event to current memory bin
5 Update write pointer
```

Function writeRegionLeaveEvent

```
1 AssignMemory( event class, call stack pointer )
2 if time leave − time of last enter ≤ ϵ then
3 |   Reset write pointer to position before enter
4 else
5 |   Write event to current memory bin
6 |   Update write pointer
7 end
8 Decrement call stack pointer
```

Figure 4.14: Algorithms to alter the Hierarchical Memory Buffer for region enter and region leave events with modifications to skip regions shorter than a minimum duration ϵ (red).

4.5 Analysis Techniques for the Hierarchical Memory Buffer

Next to construction and event reduction, methods to read and analyze the stored events form the third part of essential operations with the Hierarchical Memory Buffer. This section introduces the basic methods for linear time traversal and time stamp search; and advanced methods for statistical summaries, timeline visualization, and message matching.

4.5.1 Linear Time Iterator

A linear iterator is any object with the ability to navigate through the elements of a container object. In addition, any operation applied to an iterator does not modify the container object that is referenced. In the context of event tracing, the container object is the memory buffer holding the stored events. Most analysis methods, including all methods presented in this section, require only the basic operators

increment and *dereference*. Thus a forward iterator is sufficient for all presented analyses. At the end of this section, the operators required to form a random access iterator, are discussed, as well.

The dereference operator for the Hierarchical Memory Buffer is, similar to most other event tracing libraries, implemented in the form of a callback function [WM04, KBB+06, EWG+12]. For every event type there is a user-defined callback that provides all parameters stored with the current event. The increment operator for binary event data is usually implemented as a read of the current event to get its size and the move of the memory pointer to the position of the next event. Typically, both operators are combined to an atomic *read event* operation; which is a dereference followed by an increment.

For continuous memory buffers the increment operation is realized as described above because event data for each location is ordered by the occurrence. However, the Hierarchical Memory Buffer orders events not only by their order of occurrence but also by their hierarchy dimension and hierarchy level, i.e., their event class and calling depth. While all events of a single hierarchy partition (a single hierarchy level within a single hierarchy dimension) are ordered by their occurrence, there is no global order by occurrence in the Hierarchical Memory Buffer.

While the original order of events is lost, each event carries a time stamp, which can be used to restore the original order. Let $\prec$ be the binary comparator for the order of two events, that describes the *happened before* relation of two events, i.e., $a \prec b$ means that event a happened before event b. Furthermore, let t_i be the time stamp of event i. This implies that:

$$a \prec b \Longleftrightarrow t_a < t_b$$

Most event tracing libraries store the events in the order they occur. Some event tracing libraries strictly enforce this order within the event stream [KBB+06, EWG+12]. With $\lhd$ describing the order of two events within an event stream, the previous statement can be extended to:

$$a \prec b \Longleftrightarrow a \lhd b \Longleftrightarrow t_a < t_b$$

A single runtime event, however, may result in multiple event records with the same time stamp, e.g., entering a message send function results in an enter event for the code region, an message send event, and potential metric events. In that case, there is no happened before relation for these events and for each two events a and b that *happened together*, $a \lhd b$ and $b \lhd a$ are both valid orders in the event stream. Many monitoring environments consistently use a fixed order to store two events that happened together.

As a result, an event stream may contain events with the same time stamp and, therefore, it is not possible to restore the original order in the event stream based on time stamps. But, since both $a \lhd b$ and $b \lhd a$ are valid orders for events with the same time stamp, it is not necessary to restore the original order $\lhd$ in the event stream for events that happened together. Or in other words, an order by time stamp $<$ represents a valid order in respective to the original happened before relation $\prec$, regardless whether this order is the same order $\lhd$ in the original event stream or not. Therefore, the order by time stamp restores the original order of runtime events.

To enable an increment based on the temporal order, the Hierarchical Memory Buffer uses the time stamp order and $<$ as compare operator. The order is realized with an *event queue* that is implemented as binary minimum heap. The event queue is initialized with the first event of every non-empty hierarchy partition as shown in the algorithm in Figure 4.15.

```
Function initEventQueue
  Result: Event queue with minimum heap property containing the first event of each
          non-empty hierarchy dimension and level
1 foreach hierarchy dimension do
2     foreach hierarchy level do
3         try to read first event from hierarchical buffer at current hierarchy dimension and level
4         if success then
5             add event to event queue
6         end
7     end
8 end
9 heapify event queue
```

Figure 4.15: Algorithm to initialize the event queue.

With the minimum heap property[2] the first element in the event queue is always the event with the smallest time stamp. The atomic read event operation for the Hierarchical Memory Buffer dereferences the first element in the event queue and replaces it with the next event from the same hierarchy partition. If there is no next event the root element is removed. In either case the event queue re-establishes the minimum heap property. Figure 4.16 illustrates the basic algorithm for the read event operation.

```
Function readEvent( eventQueue )
  Data: Event queue with minimum heap property that is not empty
  Result: Event queue with minimum heap property with root replaced by next event
1 event = eventQueue.first
2 try to read next event from Hierarchical Memory Buffer at same hierarchy dimension and level
3 if success then
4     eventQueue.first = next event
5 else
6     eventQueue.first = eventQueue.last
7     remove eventQueue.last
8 end
9 shift down( eventQueue.first )
10 trigger callback for event
```

Figure 4.16: Algorithm to read next event.

As stated above, the forward iterator is sufficient for the majority of analysis methods, still it can be extended to a random access iterator. Next to the dereference and increment operators, a random access iterator needs to support the decrement, offset and direct access operators.

Getting the start position of a previous event directly is not possible due to a different number and size of parameters for each event and the additional encoding optimizations. Decrementing the iterator requires a position table containing the positions of all previous events, which can be obtained by reading all events from the beginning and storing their positions. While the creation of the position table is quite expensive, it is only necessary to create the table once. The offset operator can be realized by repeating

[2]The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

the operations increment and decrement. The direct access operator can be realized by applying the offset operator to the first event. The complexity of the basic operators for the time iterator are shown in Table 4.1 with n as the number of events and p as the number of hierarchy partitions.

Operator		Flat continuous buffer	Hierarchical Memory Buffer
dereference	*iter	$\mathcal{O}(1)$	$\mathcal{O}(1)$
increment	iter++	$\mathcal{O}(1)$	$\mathcal{O}(\log p)$
decrement	iter--	$\mathcal{O}(n)$	$\mathcal{O}(n \log p)$
offset	iter + ω	$\mathcal{O}(\omega)$	$\mathcal{O}(\omega \log p)$
offset	iter - ω	$\mathcal{O}(n)$	$\mathcal{O}(n \log p)$
direct access	iter[ω]	$\mathcal{O}(\omega)$	$\mathcal{O}(\omega \log p)$

Table 4.1: Complexity of basic operators for the time iterator.

Table 4.1 shows that the Hierarchical Memory Buffer adds a factor of $\log p$ to each operater, except for the dereferencing, caused by the shift down operation in the read event algorithm (see Figure 4.16). While the complexity of the shift down operation is in $\mathcal{O}(p)$, the actual number of shift down steps depends on the access pattern to the different hierarchy partitions. In all reviewed applications and kernels the access to the vast majority of events is restricted to neighboring partitions, which results in only a single shift down step. Section 5.3.4 shows that for all reviewed applications and kernels the cost for the shift down operations is nearly constant and, therefore, the additional factor $\log p$ is not applicable.

4.5.2 Forward Traversal

The forward traversal of event data is the most basic, as well as the most used, event data evaluation method. It reads every event of a single location and provides their values. Therefore, more complex analysis methods typically are build on top of a linear forward traversal.

The forward traversal by time for each location can be realized by repeating the *read event* operation until the end of the event data. For the Hierarchical Memory Buffer the resulting time complexity for n events and p hierarchy partitions is $\mathcal{O}(p + n \log p)$; which includes the complexity for the initialization of the event queue $\mathcal{O}(p)$ and n times the read event operation.

Another basic evaluation methods, the time stamp search operation to find the first event whose time stamp is greater than or equal to a given time stamp, can by derived from the forward traversal by repeating the *skip event* operation, which is a read event operation without triggering the callback. Thus, its complexity is analogue to the forward traversal $\mathcal{O}(p + n \log p)$.

In addition to a traversal over all events, the hierarchy information can be used to traverse only events that match a given hierarchy criterion, e.g., only communication events. Those events are a subset of all events within each location and, therefore, the number of events matching a hierarchy criterion n_h is less than or equal to the number of all events. For the Hierarchical Memory Buffer all events that match a given hierarchy criterion are within the hierarchy partitions that match the criterion. Hence, a hierarchical forward traversal can be realized with the read event operation from Figure 4.16 and a modification for the initialization of the event queue (Figure 4.17). By restricting the forward traversal to events of a subset of hierarchy partitions the complexity can be reduced to $\mathcal{O}(p_h + n_h \log p_h)$.

```
Function initEventQueueHierarchy( hierarchyCriterion )
    Data: Hierarchy criterion that defines the hierarchy partitions.
    Result: Event queue with minimum heap property containing the first event of each
            non-empty hierarchy dimension and level that match the hierarchy criterion
1 foreach hierarchy dimension ∈ hierarchyCriterion do
2     foreach hierarchy level ∈ hierarchyCriterion do
3         try to read first event from hierarchical buffer at current hierarchy dimension and level
4         if success then
5             add event to event queue
6         end
7     end
8 end
9 heapify event queue
```

Figure 4.17: Algorithm to initialize event queue with additional hierarchy criterion.

4.5.3 Statistical Summaries

Statistical summaries provide aggregated information for certain application properties for a given time interval and a given set of locations. Typical summaries include the execution time or the number of invocations of a code region, communication count or volume, and the alteration of hardware performance counters. Statistical summaries provide a coarse overview of an application's behavior. They can be used, for instance, to identify code regions that consume the most time or static load imbalances between locations.

Statistical summaries require the processing of all events that contribute to the summary for the given time interval and set of locations. Without loss of generality, in the following the time interval covers the entire run time and the set of locations is restricted to a single location. Nonetheless, all summaries can be computed in parallel for each location. The events that contribute to the summary N_s are a subset of all events N and their number $n_s = |N_s|$ is less or equal the total number of events $n = |N|$. For instance, to aggregate communication information like message count or bytes transferred, only message events need to be processes. In this case, n_s is much smaller than n (see also Figure 3.9). In continuous memory buffers these events are intermingled with all other events, which requires the processing of all events and is in $\mathcal{O}(n)$.

Within the Hierarchical Memory Buffer the additional hierarchy information can be used to process only those events in that hierarchy partitions that enclose all events that contribute to the summary. In that case, the forward traversal restricted to a subset of hierarchy partitions can be used. For all summary queries it is $N_s \subseteq N_h \subseteq N$, with N_s the set of events that contribute to the summary query, N_h the set of events in that hierarchy partitions that enclose all events that contribute to the summary query, and N the set of all events. With $n = |N|$, $n_h = |N_h|$, and $n_s = |N_s|$ it is $n_s \leq n_h \leq n$. In the majority of use cases is N_h a genuine subset of N. Therefore, the complexity for a summary query is actually reduced to $\mathcal{O}(p_h + n_h \log p_h)$.

4.5.4 Timeline Visualisation

A timeline visualization generates a visual representation of the application behavior over time. One of the most common representations, such as in Vampir [NAW+96], represents the active code region over time for each location on the horizontal axis and the selected locations on the vertical axis. Whereas each code region or group of code regions is marked with a rectangle of specific color for the time its active (see also Section 2.3).

For such a time visualization, the position of each colored rectangle, representing the active code region, must be computed. For that, the event set of each location within the current view interval needs to be processed to translate each event to its pixel position relative to the time interval. Typically, a time visualization initially shows the entire time scope and all locations, which requires to process all events of each location. The computational effort for the Hierarchical Memory Buffer is one complete forward traversal of each event stream whose complexity is $\mathcal{O}(p + n \log p)$.

However, an event stream can easily contain billions of events, while a typical display only captures 1000 to 2000 pixels in the horizontal direction. Thus, for a single information represented in each pixel millions of events are processes, which leads to an unnecessary delay of the timeline visualization. To minimize the number of processed events the forward traversal can be restricted to code region events of the lowest calling depths. For the code region events each hierarchy level contains a subset of the complete set of region events: $E_r^i \subseteq E_r \subseteq E$, with E_r the set of region events and E_r^i the set of region events of calling depth i. Since each region event has exactly one calling depth (uniqueness property) the set of region events is $E_r = \dot{\bigcup} E_r^i$ and the number of region events $n_r = |N_r| = \sum_i n_r^i$.

Thus, a timeline processing can start with the first call stack level and than iteratively add the next calling depth until there are enough events to represent the active code region for each pixel. For a fixed number of horizontal pixels ω and a variable over-determination of events per pixel σ the iterative forward traversal on increasing call stack level is finished after α call stack levels if $n_\alpha = \sum_{i=1}^{\alpha} n_r^i \geq \omega\sigma$. With this approach the complexity for the timeline visualization can be reduced to $\mathcal{O}(p_\alpha + n_\alpha \log p_\alpha)$.

4.5.5 Message Matching

A correct communication analysis for parallel applications using the message passing paradigm requires an accurate restoration of the communication from the recorded communication events. The basic operations of the Message Passing Interface (MPI), the de-facto standard for the message passing paradigm, are the point-to-point communication operations *send* and *receive*. Typically, the send and receive events are recorded separately on different locations and, therefore, the according send and receive events must be properly matched. Furthermore, all events that belong to a collective communication operation must be matched as well. Section 4.6 describes the methods for message matching in more detail, in particular, it describes an approach to handle incomplete communication traces, which might be the result of the event reduction strategies.

For the communication analysis and the message matching, especially, all MPI communication events need to be processed. Within the Hierarchical Memory Buffer those events are separated from other events. Hence, the forward traversal can be restricted to the subset of communication events $E_c \subseteq E$. In the majority of cases, E_c is a genuine subset of E and $n_c = |E_c| \ll n = |E|$. Thus, the complexity for the forward traversal can be reduced to $\mathcal{O}(p_c + n_c \log p_c)$.

Other Analysis Techniques

Next to the event trace evaluation methods presented above, there are approaches for *automatic* event trace analysis, for instance, in Scalasca [GWW+10]. In contrast to manual analysis based on an aggregated or visual representation of the application behavior, automatic analysis techniques search for patterns of typical performance problems such as load imbalances or wait states caused by unsynchronized communication (see also Section 2.3). Although, automatic analysis approaches differ from statistical summaries and timeline visualizations, they are built on a partial or total traversal of the event streams or require derived methods like statistical summaries and message matching. Similar to the presented evaluation methods, automatic analysis queries can benefit from a reduced set of events to evaluate based on additional hierarchy information.

In general, the basic evaluation operation, the forward traversal of an complete event stream, is dependent on the total number of events n in the stream, the total number of hierarchy partitions p, and the number of active (non-empty) partitions. The complexity of the forward traversal is $\mathcal{O}(p + n \log p_a)$. Whenever analysis queries require only a subset of events $E_s \subset E$, the additional hierarchy information within the Hierarchical Memory Buffer can be used to restrict the forward traversal to the events within those hierarchy partitions, so that the set of events within those hierarchy partitions E_h is a superset to E_s and it is $E_s \subseteq E_h \subset E$. In this case, the forward traversal is dependent on the number of events in those partitions $n_h = |E_h|$ and the number of according hierarchy partitions p_h. As a result the complexity of the analysis query is reduced to $\mathcal{O}(p_h + n_h \log p_{h,a})$. Section 5.3 shows a detailed evaluation and comparison of the Hierarchical Memory Buffer.

4.6 Message Matching on Incomplete Communication Data

A correct communication analysis requires the correct matching of the send and receive call of each message and in the same way a correct matching of all participating calls to a collective operation. Such a matching can be done either by a replay of the communication based on the recorded MPI events [GWW+10] or by using the order of the according events [NAW+96]. Both approaches rely on the implicitly given order of message events. Consequently, if one send or receive event is missing the correct matching of send and receive events and, therefore, the communication analysis fails. However, a correct communication analysis is essential to understand complex application behavior and identify performance issuess originating in communication. In addition, all performance metrics derived from MPI events like latency or bandwidth rely on a correct matching, as well.

This section presents a way to circumvent the unsatisfying restriction that event-based monitoring risks the entire communication analysis by dropping only a single message event. A unique sequential message identifier makes message event distinguishable from others during runtime. With such an identifier it is possible to determine which message events are missing and, therefore, it is possible to correctly match the remaining message events.

4.6.1 Message Matching Approaches

The Message Passing Interface (MPI) [MPI12] provides a set of routines to exchange data and information between different processes. The basic operations of MPI are the point-to-point communication operations *send* and *receive*. Both operations are provided in blocking and non-blocking variants.

Whereas the calls to the blocking operations return when the send or receive operations are finished and the non-blocking operations return as soon as the send or receive operations are issued. In the latter case, *test* and *wait* routines are used to check whether a message is actually finished. In both cases, a finished operation does not necessarily mean that the message was completely transferred but that the own side of the communication is finished, i.e., the message was either completely copied to an MPI send buffer or was sent via the network on the side of the sender. Built on top of these basic operations there are various *collective operations* that include all processes of a given communicator, e.g., a broad cast, scatter and gather operations for vectors, or a barrier to synchronize all processes. In this context, a communicator is a group of processes, e.g., *MPI_COMM_WORLD* containing all processes of an application. Within such a communicator each process has an unique identifier called *rank*. For all these operations users have to provide information about the message itself (a buffer containing the message and the number and data type of elements in the buffer), a communication partner, a message tag, and the communicator (see Listing 4.1).

Listing 4.1: MPI send and receive operations.

```
MPI_Send( buf, count, datatype, dest, tag, comm );
MPI_Recv( buf, count, datatype, source, tag, comm, status );
```

Each send and receive call contains a so called *message envelope* that consists of the triple communication partner, message tag, and communicator. This message envelope is used to distinguish different messages and selectively receive them. Unless communication partner, message tag and communicator match, the communication cannot be satisfied [MPI12]. In addition to the envelope, MPI uses another important criteria to identify corresponding send and receive calls. MPI guarantees that messages are non-overtaking [MPI12]. Thus, whenever multiple messages use an identical message envelope they are received in the same order they were sent.

Any message matching strategy used for a communication analysis depends on the same properties to match send and receive calls: the message envelope and the order of the messages. Figure 4.18 shows the message matching for a exemplary situation: process *P1* sends three messages with the same message envelope to process *P2*. The message matching relies on the given order of the communication events to match send and receive events, i.e., the first send event is matched with the first receive event, the second send event with the second receive event, and the same applies for the third send and receive event.

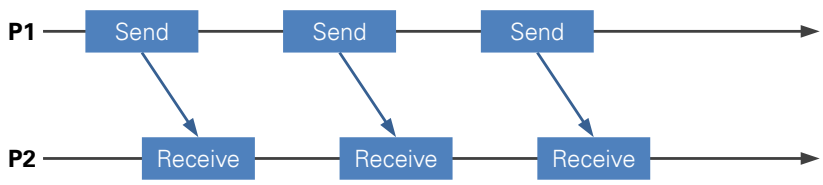


Figure 4.18: A communication pattern of three successive MPI send calls on process *P1* and the according receive calls on process *P2*; all using the same message envelope (communication partner, message tag, communicator). The according send and receive calls are matched based on their order of occurrence.

Consequently, if an MPI send or receive event is not contained in the event trace, due to filtering or event reduction, the matching of these and all successive send and receive events with the same message envelope fails or is incorrect. Figure 4.19 shows the same exemplary event sequence where the second receive event is not recorded. As a result, the message matching delivers a wrong matching: The second send event is matched with the originally third receive event and the third send event cannot be matched. While the unmatched send event can be at least detected, the mismatched second send event might remain unrevealed. In particular, if the third send event had not been recorded as well, the message matching would not discover any error in the matching and proclaim the message matching as successful and correct. In the given example, even if the message matching discovers the missing receive event, it is not possible to identify which receive event is missing. Therefore, the complete matching of all events with this message envelope must be considered incorrect.

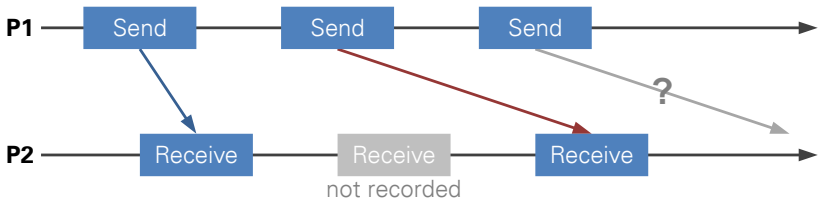


Figure 4.19: The second receive call on process *P2* is not recorded. As a result, the original order of the events is modified and the message matching of this event sequence fails.

4.6.2 Identification of Missing Communication Events

To enable a correct message matching even on incomplete MPI event data, each message needs to be distinguishable from other messages. Therefore, whenever multiple messages use the same message envelope (partner, tag, communicator) an unique identifier as fourth parameter, replacing the indirectly given order, is necessary to explicitly identify each message. Obviously, such an identifier must be stored with both, the send event as well as the according receive event.

Piggybacking techniques on top of MPI can transport such an identifier from the sender to the receiver with every message that is send. Three piggybacking techniques are studied in [SBS08]. The first approach uses additional messages, i.e., send a second message after each actual message. This approach is problematic for performance monitoring of the communication itself because of the latency and bandwidth overhead. In addition, this approach fails for wildcard[3] receives. The other two approaches create new MPI data types by copying the original message and additional information in a separate buffer. Both approaches introduce very high overhead, as well. While piggybacking is useful for analyses that are not critical in terms of time, e.g., error detection, piggybacking cannot be applied for performance monitoring as long as MPI does not provide a low overhead piggybacking mechanism itself [WDKN13].

The Sequential Message Identifier

Therefore, this thesis introduces a *sequential message identifier* σ that generates the same unique identifier for the sender and receiver of a message without the need to transfer additional data via MPI. Such an sequential message identifier enumerates all messages of each message envelope separately. During

[3]MPI allows the use of wildcards in receive calls: MPI_ANY_SOURCE to accept a message from an arbitrary sender and MPI_ANY_TAG to receive a message with an arbitrary message tag.

runtime, this approach identifies the message envelope for each message, increments the sequential message identifier, and stores its current value with the according send and receive events. For each message envelope an internal management data structure keeps an entry that stores the message envelope and the sequential message identifier. One prerequisite of this approach is that every MPI point-to-point message call is instrumented and captured, so the internal sequential message identifier can be incremented. This can be achieved by wrapping the MPI library via the PMPI interface, which is the default for most monitoring tools. In this way, every send and receive event is tagged with an unique identifier per message envelope that is the same for the according send and receive of each message. Listing 4.2 shows the extension to the existing send event record in OTF2 [EWG+12] to store the sequential message identifier; in the same way the receive record and the non-blocking variants can be adapted.

Listing 4.2: OTF2 MPI send event record with a 4-byte sequential message identifier.

```
typedef struct OTF2_MpiSend
{
   uint64_t time;
   uint32_t receiver;
   uint32_t communicator;
   uint32_t msg_tag;
   uint64_t msg_length;
   uint32_t sequence_id; /* sequential message identifier */
};
```

4.6.3 Adapted Message Matching

A message matching technique needs to incorporate this sequential message identifier, as well. Instead of the message envelope alone, the quadruple of communication partner, message tag, communicator, and sequential message identifier σ guides the decision whether a matching is correct or not. The matching is correct if and only if the corresponding send and receive events use the same message envelope and have the same sequential message identifier. Messages where only one event of the message is stored need to be ignored for matching. These message can be easily detected during the message matching: Whenever there is a gap in the sequential message identifier intermediate messages are missing. In particular, if there is a message with the sequential message identifier $\sigma = n$ and for the next message with the same message envelope the sequential message identifier $\sigma = n + m$ with $m \geq 1$, it can be concluded that the messages with the identifier $n + 1$, ..., $n + m - 1$ are not recorded. If the according send or receive event has been recorded it can be ignored for the further message matching.

Figure 4.20 illustrates the enumeration of the send and receive events with the sequential message identifier for the previous example. Each send and receive event is tagged with the according sequential message identifier. The message matching incorporates the new sequential message identifier. Since the receive call with the sequential message identifier $\sigma = 1$ on process P2 is missing, a matching to the corresponding send call with $\sigma = 1$ on process P1 is not possible. Therefore, the send event with $\sigma = 1$ is ignored in message matching process.

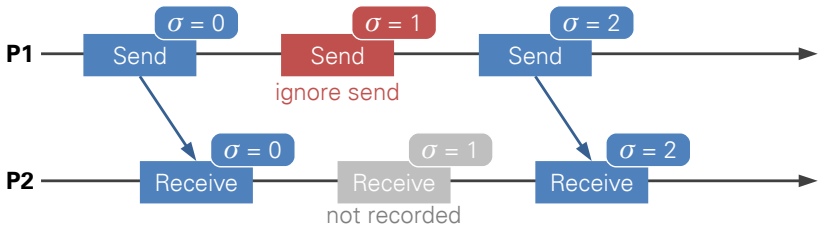


Figure 4.20: Communication pattern of three successive messages from process P1 to process P2 where the second receive call is not recorded. With the new sequential message identifier σ the missing receive event is detected and the send and receive events can be matched correctly.

There are two scenarios that require a separate treatment, which are shortly covered in the next paragraph. First, receive calls that use wildcards like MPI_ANY_SOURCE or MPI_ANY_TAG do not include the communication partner or message tag, respectively. However, the status parameter of each receive call keeps the actual sender and tag of the message that was receive with that call. Since, the receive event is stored after the actual receive of a message, the status parameter can be used to retrieve the correct message envelope and, thus, apply the correct sequential message identifier. The second case is the use of MPI_Cancel, which tries to cancel a pending non-blocking send or receive operation. In that case, an offset between the sequential message identifier of send and receive events occurs. This offset must be considered in the message matching algorithm.
Next to the basic point-to-point operations, MPI provides collective operations built on top of the basic operations. The Open Trace Format 2 already provides a so called *matching ID* for collective operations, which uniquely identifies each collective operation. This matching ID can be used in a similar way as the sequential message identifier to detect missing collective operations or, more importantly, missing partners of a collective operation.

In summary, with the introduced sequential message identifier that makes MPI events distinguishable from others during runtime, it is possible to identify missing MPI events and match the remaining MPI events correctly. With this approach and an adapted message matching technique that incorporates the sequential message identifier it is possible to apply event reduction without sacrificing a detailed communication analysis. Section 5.3.5 reviews the feasibility and introduced overhead of this approach.

4.7 Adaption to Sampling

This section adds a short excursion for a further use of the Hierarchical Memory Buffer in the context of sample-based tracing. While this thesis focusses on event tracing, periodic sampling is another method to generate trace data (see Section 2.1). Instead of runtime events, sampling approaches record the current state of an application, usually, with a fixed frequency. The current state of an application that is recorded is typically the current calling context and adequate performance metrics. Recording the state of an application with a fixed frequency provides the inherent benefit that the recorded data rate can be approximated and regulated with the sampling frequency. However, finding an optimal sampling frequency is a virtually impossible task: Using a too low sampling frequency maps the application behavior very coarse. Using a too high sampling frequency results in huge data volumes. In addition, new ap-

proaches not only refer to time intervals to record a sample but to intervals of hardware performance counters [Wea14], e.g., with every n-th cache miss or floating point operation. For those approaches it is even more difficult to find useful sampling intervals because their progress is not predictable like a frequency based on time intervals.

An hierarchical memory representation of the recorded samples supports a reduction of already recorded samples similar to events. Such a runtime reduction of samples allows to start recording with a very high sampling frequency. Whenever the recording memory buffer is exhausted every n-th sample is discarded and the sampling frequency adopted accordingly. Such an approach releases the user to find an appropriate sampling frequency on its own by automatically adapting the sampling frequency to utilize the recording memory buffer. Even more effective is such an approach for not time-based and, therefore, not predictable sampling intervals based on hardware performance counters.

Since, samples contain similar hierarchical informations like events, e.g., the calling context, this hierarchy information can be used to store samples similar to events in the different hierarchy levels, e.g., according to their call stack level. In contrast to event-based trace data, hierarchical ordering in terms of time is a much more relevant hierarchy dimension. Since samples are triggered by the sampling interval rather than application context, each sample by itself represents a randomly chosen application state. In particular, for any two individual samples it is impossible to identify which one of these better represents the actual state of an application. In addition, each sample on its own contains the complete state of an application it represents; while most events are only meaningful in combination with further events, e.g., a code region is only represented with the according enter and leave event. Thus, selecting samples for reduction by a fixed ratio based on the order of occurrence, e.g., discarding every n-th entry is a very powerful method for samples, whereas this method does not make any sense for events.

As stated before, to apply a reduction method efficiently, the entries in the Hierarchical Memory Buffer need to be sorted accordingly. Although a sample by itself does not contain sufficient hierarchy information to presort samples for a reduction based on the ratio, the order of occurrence can be used to distribute the samples to different hierarchy levels to enable an efficient reduction. The distribution function $\lambda : \mathbb{N} \rightarrow \mathbb{N}$ that maps each sample to a hierarchy level based on the order of occurrence n can be expressed as:

$$\lambda(n) = max\Big\{\ p \in \mathbb{N} \mid n \equiv 0\ mod\ 2^p\ \Big\} \tag{4.3}$$

This way each level λ contains every $2^{\lambda+1}$th sample. Figure 4.21 illustrates this mapping for the first few samples. The lowest hierarchy level λ_0 contains every 2nd sample, the second lowest hierarchy level holds every 4th sample, and so on, up to the highest hierarchy level $\lambda_{\text{max}} = \lfloor \ln n \rfloor$, which consists of only one sample. Since each level λ contains every $2^{\lambda+1}$th sample, the interval of levels $[\lambda, \infty)$ contains every 2^{λ}th sample. Therefore, the lowest hierarchy level λ_0 contains one half of the samples and the levels $[\lambda_1, \lambda_{\text{max}}]$ contain the other half of the samples.

In the reduction operation all samples on the lowest hierarchy level are discarded and the according memory bins are released. In addition, from that point the sampling frequency is divided in half. Due to the distribution based on powers of two, after the reduction operation the lowest hierarchy level λ_1 contains one half of the remaining samples and the levels $[\lambda_2, \lambda_{\text{max}}]$ contain the other half of the samples. This way, the reduction operation can be applied iteratively whenever the memory buffer is exhausted; each time discarding every 2nd sample.

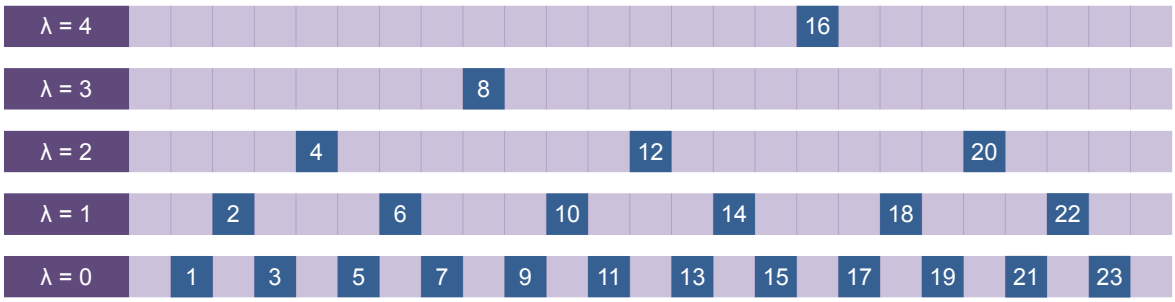


Figure 4.21: Distribution of samples based on powers of two.

However, the maximum notation of the distribution function λ in Equation 4.3 may be very compute intensive, especially, for large numbers of samples. Since any natural number n can be uniquely decomposed in two-potencies in the form of

$$n = \sum_{p \in \mathbb{N}} \alpha_p 2^p, with\ \alpha_p \in \{0, 1\}$$

the distribution function λ of Equation 4.3 can also be expressed as:

$$\lambda(n) = min\left\{ p \in \mathbb{N} \,\middle|\, n = \sum_{p \in \mathbb{N}} \alpha_p 2^p \wedge\ \alpha_p = 1 \right\} \tag{4.4}$$

This minimum notation provides a more efficient way to compute λ because the representation as two-potencies equals the binary representation of integer values. For instance,

$$\begin{aligned} 150_{10} &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 10010110_2 \end{aligned}$$

Therefore, the statement in Equation 4.4 is equal to the number of trailing zeros in a binary representation of n, which can be calculated with minimal costs.
With this distribution function based in combination with the Hierarchical Memory Buffer it is possible to automatically adapt the sampling frequency during runtime to utilize the recording memory buffer without risking a buffer overflow or a too coarse sampling frequency.

4.8 Summary

From the event reduction strategies introduced in Section 3.4 arises a need for an efficient identification and elimination of events that are already stored in the memory buffer. While all existing approaches, using a flat and continuous memory buffer, allow an application of the event reduction strategies, the enormous introduced overhead renders the benefits of an in-memory event tracing workflow futile. In a flat continuous memory event representation all events are stored in the order of their occurrence. Since all events with different hierarchy criteria such as event class and calling depth are intermingled, an event reduction operation must scan the entire memory buffer to identify those events that match the reduction criterion and mark their occupied memory segment as free. In addition, all remaining memory segments need to be collapsed to provide a free continuous memory segment at the end of the memory buffer to store further events. This results in a computation complexity of $\mathcal{O}(n)$ for n events stored in the memory buffer.

In contrast, the new Hierarchical Memory Buffer data structure enables a hierarchical event representation. Such an hierarchical event representation uses additional hierarchy information, like the event class and calling depth, to store events in a multi-dimensional array, where each array entry, called hierarchy partition, keeps its own small memory buffer and represents exactly one event class and one calling depth. Instead of one huge memory chunk, the total memory allocation of the memory buffer is divided in many small memory sections, called memory bins, that are dynamically distributed to any entry in the multi-dimensional memory buffer array. This allows a high memory balance between all hierarchy partitions and utilizes the provided memory allocation. In such a hierarchical event representation, all events are presorted according to their event class and calling depth. This way, in the Hierarchical Memory Buffer an event reduction operation includes only the revoking of all memory bins that have been distributed to those hierarchy partitions that match the criterion for event reduction. Therefore, the computational complexity for an event reduction operation is reduced to $\mathcal{O}(b)$ for b memory bins in the according hierarchy partitions.

The Hierarchical Memory Buffer itself consists of two vectors representing the two hierarchy dimensions event class and calling depth. Within the vectors each entry represents one hierarchy partition and the according memory bins are managed as single linked lists that include a pointer to the current write and read position. This allows linear write access in constant time complexity and a complete forward traversal of the memory buffer in $\mathcal{O}(n \log p)$ for n events and p non-empty hierarchy partitions. Furthermore, the additional hierarchy information supports a partial forward traversal of the memory that is restricted to a subset of events and hierarchy partitions. Such a partial traversal benefits all analysis methods that are restricted to a subset of events, for instance, the communication analysis or are step-wise composition of a timeline visualization that only processes a subset of call stack levels depending on the zoom level.

The Hierarchical Memory Buffer forms the second main contribution of this thesis. It allows to perform the aforementioned event reduction operations with minimal overhead and supports new event selection and filter operations unfeasible with a traditional flat continuos memory buffer layout. In addition, several typical analysis requests can benefit from a hierarchy-aided traversal of recorded event data. Its performance and constraints are evaluated in detail on the basis of real-life applications and application kernels in Section 5.3.

5 Evaluation and Case Study

This chapter presents an evaluation of the enhanced encoding techniques and the Hierarchical Memory Buffer data structure including its capabilities to support the event reduction strategies. In addition, a detailed case study demonstrates the benefits of the combined approach for a real-life application.

5.1 Methodology and Target Applications

A library that supports current and future performance analysis tools faces the same key challenges as discussed in Section 2.4. Therefore, this Chapter evaluates the presented optimizations from Chapter 3 and 4 based on their contribution to meet these key challenges: How they overcome current scalability bottlenecks, how they cope with the enormous created data volumes, and how they minimize the bias on the recorded application.

The evaluation utilizes theoretical models whenever possible to determine best, worst and average case behavior. In addition, the evaluation relies on empirical data obtained from synthetical benchmarks and real-life applications and application kernels to either support the theoretical models or provide results for situations that cannot be sufficiently modeled. The applications include a subset of the SPEC MPI 2007 benchmarks [MWL+07], the NAS Parallel Benchmarks [BLBS92] in version 3.3, and the real-life applications Gromacs [HKS08], COSMO-SPECS+FD4 [LGW+12], and Semtex [BS04]. The application kernels from the benchmarks and the real-life applications represent a variety of different research fields, different communication behaviors, event composition, and different length.

Since, a number of recorded application parameters varies with each recorded run, e.g., time stamps, all measurements that include real-life applications and kernels are based on the data of existing event traces. This allows a fair comparison of the different approaches that eliminates variations in application behavior and system performance. For the SPEC MPI 2007 benchmarks[1] all event traces were generated using the problem size "train" and ran on an SGI Altix 4700 system. The NAS Parallel Benchmark[2] traces use the problem size "C", except for BT, which uses problem size "B", and were recorded on an Intel Xeon cluster. Gromacs is a package to perform molecular dynamics for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions[3]. The recorded trace includes 50 iteration blocks and was recorded on an Cray XC system. COSMO-SPECS+FD4 is a model system for detailed cloud simulations that consists of a regional atmospheric model (COSMO), a detailed cloud microphysics model (SPECS), and a scalable load balancing and coupling framework (FD4). The recorded trace includes 2 simulation iterations and was recorded on an Intel Xeon cluster. Semtex is a set of spectral element simulation codes, most prominently a code for direct numerical simulation of incompressible flow. The trace was recorded on an SGI Altix 4700 system.

[1] http://www.spec.org/mpi2007/
[2] http://www.nas.nasa.gov/publications/npb.html
[3] http://www.gromacs.org

Table 5.1 gives an overview of all evaluated applications and kernels including their usual application area, the per location trace sizes[4], the runtime, the total number of events and the machines clock rate.

	Application Area	Trace Size*	Runtime	Events*	Clock Rate
104.milc	Quantum Chromodynamics	1.4 GiB	231 s	178M	1.6 GHz
107.leslie3d	Computational Fluid Dynamics	198 MiB	57 s	2.6M	1.6 GHz
115.fds4	Computational Fluid Dynamics	85 MiB	27 s	11M	1.6 GHz
121.pop2	Ocean Modeling	245 MiB	92 s	31M	1.6 GHz
122.tachyon	Parallel Ray Tracing	5.3 GiB	1620 s	665M	1.6 GHz
126.lammps	Molecular Dynamics Simulation	36 MiB	16 s	4.9M	1.6 GHz
127.wrf2	Weather Prediction	694 MiB	361 s	87M	1.6 GHz
129.tera_tf	3D Eulerian Hydrodynamics	453 MiB	154 s	60M	1.6 GHz
130.socorro	Molecular Dynamics	2.1 GiB	1808 s	259M	1.6 GHz
137.lu	Computational Fluid Dynamics	17 MiB	16 s	2.2M	1.6 GHz
BT	Block Tri-diagonal Solver	4.0 GiB	288 s	231M	2.8 GHz
CG	Conjugate Gradient	406 MiB	35 s	24M	2.8 GHz
EP	Embarrassingly Parallel	2.3 MiB	6.1 s	0.1M	2.8 GHz
FT	Discrete 3D Fast Fourier Transform	23 MiB	18 s	1.4M	2.8 GHz
IS	Integer Sort	1.1 GiB	30 s	67M	2.8 GHz
LU	Lower-Upper Gauss-Seidel Solver	134 MiB	53 s	7.9M	2.8 GHz
MG	Multi-Grid	5.1 MiB	6.6 s	0.3M	2.8 GHz
SP	Scalar Penta-diagonal Solver	142 MiB	100 s	8.7M	2.8 GHz
Gromacs	Molecular Dynamics Simulation	2.5 GiB	310 s	112M	1.0 GHz
Cosmo-Specs	Atmospheric Modeling	2.3 GiB	87 s	134M	2.8 GHz
Semtex	Computational Fluid Dynamics	17 GiB	2308 s	1024M	1.6 GHz

* Average value per location.

Table 5.1: Overview of evaluated applications and benchmarks.

The following section focusses on the evaluation of the enhanced encoding techniques presented in Section 3.3. After that, Section 5.3 reviews the Hierarchical Memory Buffer data structure introduced in Chapter 4 and its capabilities to support the event reduction strategies. Finally, Section 5.4 presents a detailed case study of the benefits of the combined approach for the Gromacs application.

[4]Trace sizes in uncompressed Open Trace Format [KBB+06].

5.2 Enhanced Encoding Techniques

The enhanced encoding techniques presented in Section 3.3 target the per location internal memory representation in the Open Trace Format 2 [EWG+12]. Thus, these techniques have no impact on OTF2's scalability but affect the consumed memory and the runtime overhead. At first, the following section focuses on the memory allocation during runtime. After that, Section 5.2.2 discusses the introduced runtime overhead.

5.2.1 Runtime Memory Allocation

The primary goal of the enhanced encoding techniques is to drastically increase memory efficiency, which leads to less allocated memory during runtime and, more importantly, less bias on the recorded application behavior due to fewer memory buffer flushes. Next to that, the overall trace size of a measurement is reduced. Since, the effect of each enhanced encoding technique is highly correlated to the actual parameters of each single event and to the outcome of the other encoding steps, it is virtually impossible to exactly model the increase of memory efficiency for each encoding technique, individually. Nonetheless, the following enumeration lists beneficial and hurtful criteria as well as a simplified model for each encoding step:

- **Splitting of timing information and event data:** The effect of this encoding step is directly correlated to the average event size and the number of duplicate time stamps, which can be omitted. Thus, the trace size without encoding is $\sum_{all\ events}(\tau + \epsilon + 1)$, with an average time stamp size τ, an average event size ϵ and one additional byte for the identifying token. The trace size with the splitting of timing information and event data is $\sum_{all\ events}(\tau_{all}+\epsilon+2)-\sum(\tau_{duplicate}+1)$, which adds an additional token byte for the time stamp event and is reduced by the size of all duplicate time stamps. Thus, in worst case, there are no duplicate time stamps with leads to an increased trace size of n bytes, with n the number of events. A good case, would provide multiple events with the same time stamp. For instance, if h hardware performance counters are stored with each region entry and exit the resulting trace size[5] is approx. $\sum_{all\ events}(\frac{1}{h}\tau + \epsilon + 2)$.
- **Leading zero elimination:** The result of the leading zero elimination depends on the average values of all parameters. High values lead to less reduction than smaller values. More precisely, not the value itself determines the reduction but its number of used bits within the integer encoding.
- **Delta encoding** decreases the values that need to be stored, which leads to a better utilization of the leading zero elimination. Therefore, parameters with small deltas and starting with a very high offset, e.g., time stamps or some hardware performance counters, benefit the most.
- **Encoding within the token** benefits the most frequent events such as time stamps, region entry/exit, and metric events. Since, all other events have only a marginal share, the average event size is reduced by approx. two bytes – one byte each for the according time stamp and the event itself – and the total trace size is reduced to $\sum_{all\ events}(\tau + \epsilon - 1)$.
- **Timer resolution reduction:** In contrast to the previous encoding techniques, a reduction of the timer resolution leads to a loss in detail. The effect of this encoding step mainly depends on the amount of timer resolution reduction.

[5] The number of events other than region entry and exit is ignored due to their negligibly small share (see also Figure 3.9).

The Effects of Each Encoding Technique

Figure 5.1 shows the effects of each encoding step for the reviews real-life applications and kernels. Since most encoding steps build upon other steps, from left to right, each encoding step includes all previous encoding steps. In addition, Table 5.2 lists the minimum, average, and maximum reduction of memory allocation for each encoding technique relative to the previous encoding step. The results from the SPEC MPI 2007 application kernels include one metric event (floating point operations hardware performance counter) for each region enter and leave event; all others do not include any metric events. Therefore, splitting timing information and event data decreases memory allocation for the SPEC MPI 2007 application kernels while for all others the memory allocation is slightly increased.

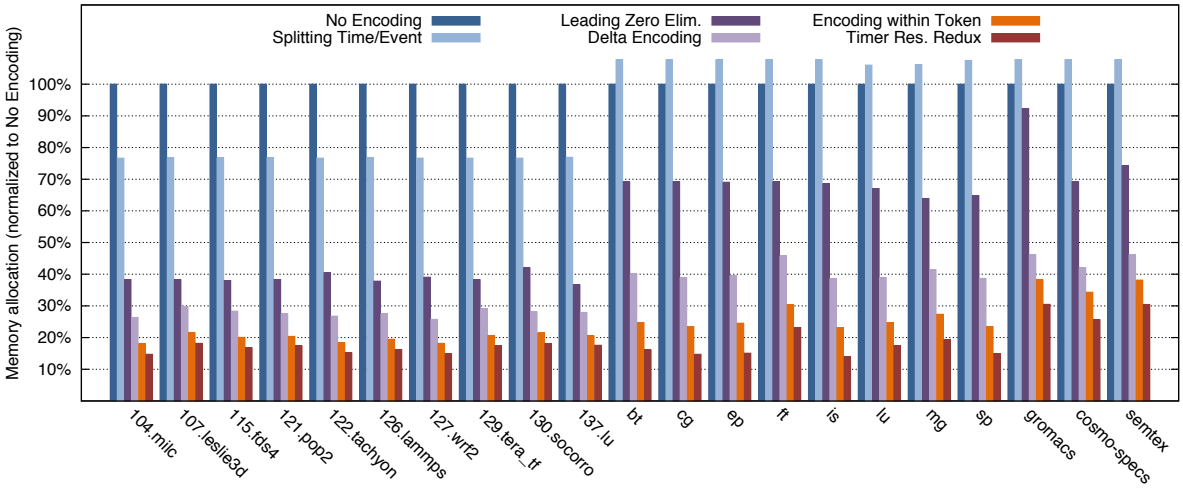


Figure 5.1: Memory allocation for different encoding techniques.

		Minimum	**Average**	Maximum
Splitting of timing information and event data	one metric	23.1 %	**23.3 %**	23.3 %
	no metrics	-7.6 %	**-7.4 %**	-6.0 %
Leading zero elimination		14.3 %	**41.5 %**	52.3 %
Delta Encoding		22.2 %	**34.9 %**	50.0 %
Encoding within the token		17.2 %	**30.2 %**	39.9 %
Timer resolution reduction		14.4 %	**23.7 %**	39.9 %

Table 5.2: Reduction of memory allocation for each encoding technique relative to the previous step.

From Figure 5.1 it can be inferred that, first, splitting timing information and event data is only effective when metric records are recorded frequently, e.g., with every region enter and exit. Second, the leading zero elimination combined with delta encoding drastically reduces the memory allocation. In particular, the results for Gromacs show a noticeable weaker decrease for the leading zero elimination (due to high starting offsets). But combined with delta encoding the memory allocation is in the same range as the other applications. Third, the encoding of values within the token, especially for the region enter and exit events, results in a weaker effect for the three real-life applications since there is a considerably higher count of different regions (see also Figure 3.12). Fourth, the reduction of the timer resolution can further

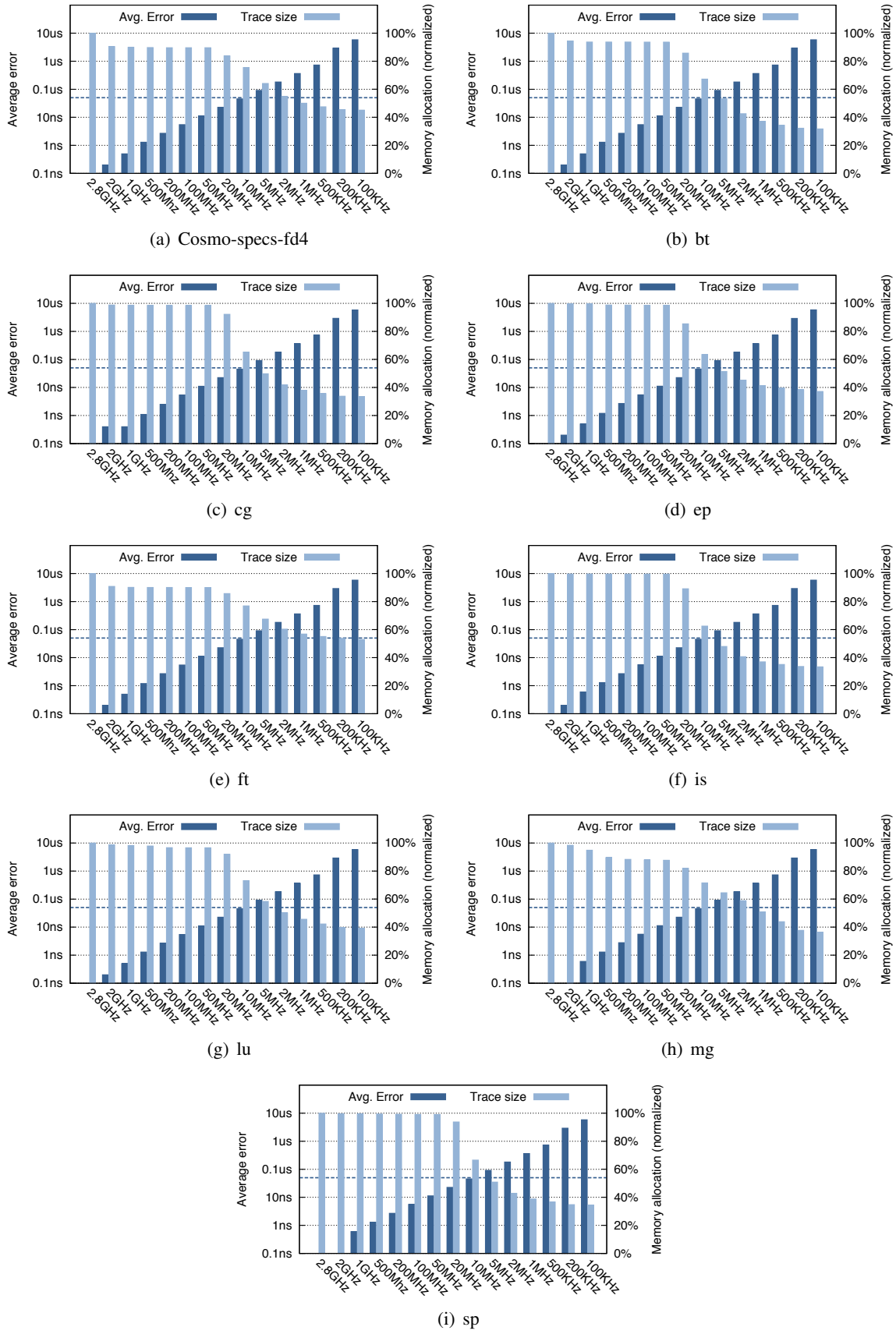
decrease the memory allocation. However, this encoding step introduces a loss in accuracy that depends on the magnitude of the resolution coarsening. The results in Figure 5.1 use a timer resolution of 10 μs. Finding a good trade-off between memory allocation and accuracy is evaluated next. At any rate, the encoding techniques proof to be effective and reduce the memory allocation during runtime to a fraction of 13 - 30 % for the reviewed applications and kernels.

Timer Resolution Reduction

The timer resolution reduction can provide a further decrease of runtime memory allocation at the cost of loosing fine detailed timing information. To determine a timer resolution that serves both, a small memory allocation and a justified bias, it is important to analyze the accuracy of a single high resolution time stamp. Given that the total overhead of the measurement environment on the observed application of 5 % is considered acceptable implies that an average error of each time stamp in the range of 5 % is likewise acceptable. Figure 3.13 shows that the average event frequency is below one event per microsecond. Based on this average event frequency an acceptable error of 5 % means an average error of 50 nanoseconds per event.

Figure 5.2 demonstrates the effect of different reduced timer resolutions for the applications and kernels with the highest original timer resolution. It depicts allocated memory in light blue and the average error per event in dark blue (in logarithmic scale). In addition, the blue dotted line represents the 50 nanoseconds error range. For all applications in each case the average error is about half of the timer resolution, which is trivial since it is basically just the round-off error. Thus, for a timer resolution of 10 MHz, i.e., rounding down to hundreds of nanoseconds, the error is about 50 nanoseconds. In addition, Figure 5.2 shows that a 10 MHz timer resolution is also a sweet-spot in terms of memory allocation because in most cases the sharpest drop is from 20 MHz to 10 MHz.

Consequently, if a timer resolution reduction is applied to further decrease runtime memory allocation, a reduction to a timer resolution of 10 MHz provides the best trade-off between memory allocation and accuracy.



(a) Cosmo-specs-fd4

(b) bt

(c) cg

(d) ep

(e) ft

(f) is

(g) lu

(h) mg

(i) sp

Figure 5.2: Average time error per event and allocated trace size for different reduced timer resolutions.

Comparison With Other Event Tracing Formats

The following measurements compare the prototype, called OTFX, which is based on OTF2 [EWG+12] and includes all presented enhanced encoding techniques, with other well-established trace formats and libraries. However, comparing different trace formats is a non-trivial task. In most cases, the trace libraries are closely integrated in the according event tracing tools, which generate their own individual overhead. Most importantly, the different tracing tools and trace data formats do not store exactly the same information. They focus on different application behavior characteristics, different levels of detail, and provide a different amount of functionality. Therefore, it is necessary to decouple the trace data libraries from the tracing tools to eliminate all effects that originated in the tracing tools and not the data formats. By using the data of existing application traces, as described above, it is ensured that all formats store exactly the same information, which is crucial when comparing the encoding techniques.

With this in mind, the comparison considers only those trace data formats that store comparable data. The first is the EPILOG trace format [WM04] used by the SCALASCA tool set [GWW+10], which uses binary encoding. The second is the Open Trace Format (OTF) [KBB+06] employed by the Vampir tool set [NAW+96] using ASCII encoding. Since the according VampirTrace measurement infrastructure uses its own internal memory buffering based on aligned C data structures, this is also taken into account for the comparison. In fact, the OTF data format is only used to store data to the file system. Nonetheless, it provides an efficient encoding that is reviewed, as well. The third format is the Open Trace Format 2 [EWG+12] used by the Score-P measurement infrastructure [KRM+12], which is based on binary encoding and already applies the splitting of timing information and partly the leading zero elimination developed in this thesis. These data formats also describe the starting point of the encoding techniques presented in this thesis, except for OTF2 that already includes some first optimizations presented here.

The trace data formats are compared to the OTFX prototype excluding the timer resolution reduction to keep the information exactly the same. In this comparison the enhanced encoding techniques deployed in the OTFX prototype show a remarkable decrease in memory allocation. OTFX consumes about 91 % less memory than VampirTrace, about 79 % less memory than OTF, about 71 % less memory than EPILOG, and about 70 % less memory than OTF2 (see Figure 5.3).

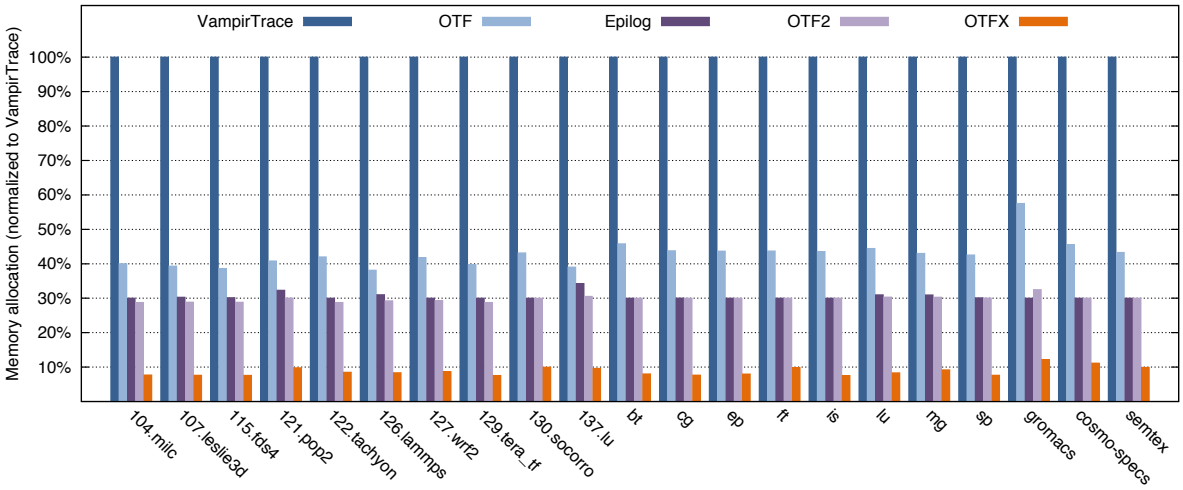


Figure 5.3: Memory allocation for different event trace data formats.

Comparison With General Purpose Compression

To classify the capabilities of the encoding enhancements, they are also compared to the Open Trace Format 2 with the compression from zlib [DG96], which is a well-established compression library based on the Lempel-Ziv 77 compression algorithm [ZL77] and provides a good trade-off between compression ratio and overhead [SL11]. Figure 5.4 shows the memory allocation for OTF, OTF2 with and without zlib compression, and OTFX with and without timer resolution reduction (trr). The enhanced encoding techniques achieve a slightly lower decrease in memory allocation than zlib (74 %) without timer resolution reduction (70 %) and a slightly higher decrease in memory allocation with timer resolution reduction (77 %, all relative to OTF2). Thus, the enhanced encoding techniques realize an equal memory efficiency as general purpose compression of the LZ77 family.

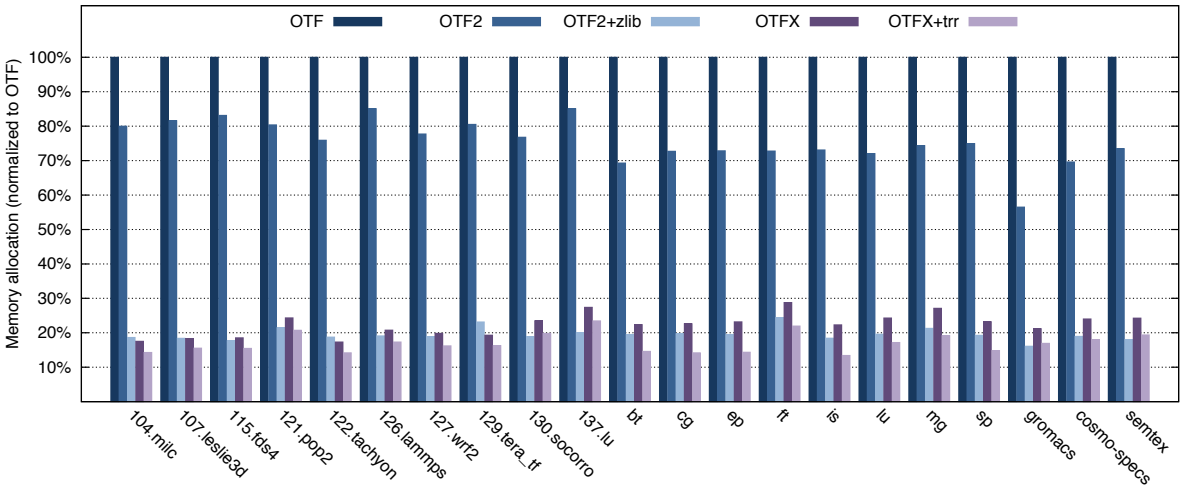


Figure 5.4: Memory allocation in detail for OTF, OTF2 including zlib compression, and OTFX with and without timer resolution reduction (trr).

5.2.2 Runtime Overhead

Next to memory efficiency, the introduced runtime overhead is equally important for a successful measurement of an application. Adding too much overhead will drastically reduce the accuracy of a measurement and bias the recorded application behavior. This means, an event trace format as part of an event tracing tool has to introduce as less overhead as possible. Therefore, improvements in memory efficiency must be judged by their overhead, as well.

Figure 5.5 shows the runtime overhead of OTF, OTF2, OTF2 with zlib compression, and OTFX including the timer resolution reduction. To exclude file system overhead, the measurements only include the storage within the memory during runtime. Hence, some applications are not included in the figure because their collected data exceeds all memory capabilities.

Two important results can be drawn from Figure 5.5. First, the enhanced encoding techniques do not introduce additional overhead. The overhead is in the same range as OTF2 without the enhanced encoding. Second, while Figure 5.4 might encourage to use a general purpose compression library to increase memory efficiency, this will drastically raise the runtime overhead. Even more, overhead introduced by a compression library is usually not equally distributed over all events. In most cases, a certain amount

of data is collected and then compressed within one step to achieve a high compression ratio. In this measurements, compressing every 1 MiB creates an overhead of 26 - 29 ms each. Therefore, using a compression library for higher memory efficiency is unfavorable. The results of Epilog and VampirTrace are not included but have already been covered in [EWG+12]: The overhead of Epilog is comparable to OTF2. The overhead of VampirTrace's internal memory buffering is about 2.5 times lower than OTF2 since all event attributes are aligned in memory, however, at the price of significantly increased memory requirements due to the padding.

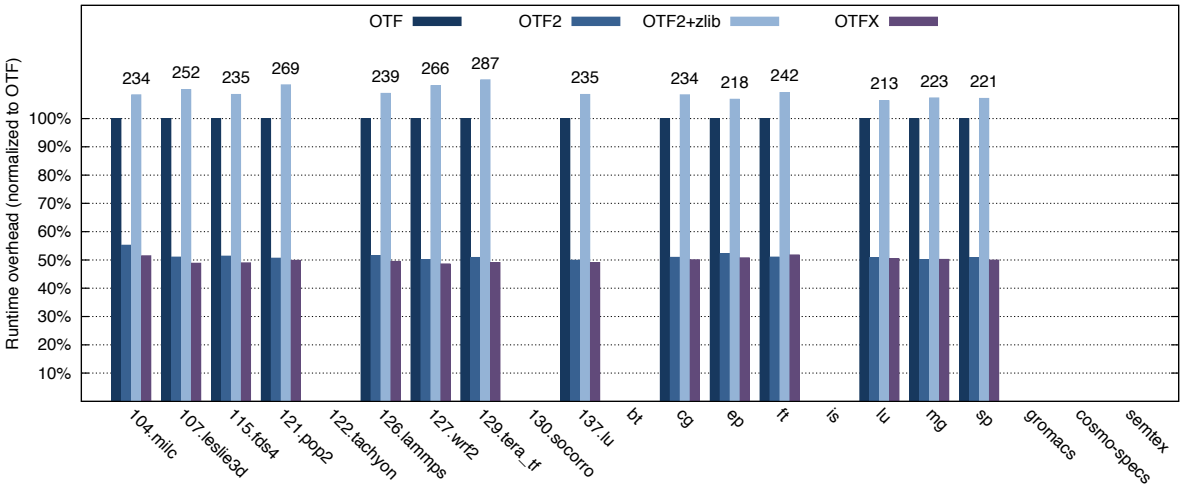


Figure 5.5: Runtime overhead of OTF, OTF2 with and without zlib compression, and OTFX.

The measurements demonstrate that the enhanced encoding techniques discussed in Section 3.3 remarkably reduce the memory allocation for event trace data during runtime without increasing the overhead of the tracing library. The results of this section are summarized in Table 5.3, which shows the runtime overhead and the runtime memory allocation in comparison to OTF. The OTFX prototype that includes all presented encoding techniques consumes about 80 % less memory without and about 83 % less memory with timer resolution reduction compared to OTF. With this, OTFX achieves the memory efficiency of well-established compression libraries without introducing their respective overhead, which is about five times higher than with OTFX.

Trace Format	Runtime Overhead			Memory Allocation		
	Minimum	Average	Maximum	Minimum	Average	Maximum
OTF		**100.0** %			**100.0** %	
OTF2	49.9 %	**51.2** %	55.2 %	56.5 %	**75.6** %	85.1 %
OTF2 with zlib	212.7 %	**240.7** %	286.9 %	16.1 %	**19.5** %	24.4 %
OTFX	48.7 %	**50.3** %	52.3 %	17.3 %	**22.4** %	28.8 %
OTFX with trr	48.6 %	**49.9** %	51.8 %	13.4 %	**17.0** %	23.5 %

Table 5.3: Runtime overhead and memory allocation of OTF, OTF2, OTF2 with zlib, and OTFX. All values are normalized to OTF's overhead or memory allocation, respectively.

5.3 The Hierarchical Memory Buffer

This section evaluates the Hierarchical Memory Buffer data structure introduced in Chapter 4 and its capabilities to support the event reduction strategies. While current approaches rely on a flat continuous memory segment, the new Hierarchical Memory Buffer uses a hierarchical ordering and many small memory segments that are distributed on demand. Similar to the enhanced encoding techniques, the altered memory layout influences the per location internal memory representation. Hence, it only affects the consumed memory and the runtime overhead but not the event tracing library's scalability.

The dominant parameter influencing both criteria is the size of the dynamically distributed memory segments, called memory bins. The following section aims to find a suitable size for these memory bins. After that, Section 5.3.2 evaluates the efficiency of the new data structure for event reduction.

5.3.1 Determine an Ideal Memory Bin Size

Choosing a feasible size for the memory bins is crucial. On the one hand, memory efficiency decreases with bigger memory bins because the memory bins may often not be fully utilized. For instance, writing only a few events to each call stack level will result in a much bigger total memory allocation for bigger memory bins than for small ones since each way there is one memory bin allocated for each hierarchy partition[6]. Moreover, in the multi-dimensional layout of the Hierarchical Memory Buffer a lot of memory bins are necessary to distribute at least one memory bin to each non-empty hierarchy partition. Otherwise, reduction is triggered way to early because the Hierarchical Memory Buffer runs out of memory bins; not because their memory is exhausted. Therefore, it is desirable to keep the size of the memory bins as small as possible. On the other hand, with smaller memory bins, the overhead introduced by managing the memory bins is expected to increase, since there are more memory bins to allocate and operate. But, from a technical point of view, there is another disadvantage of smaller memory bins. Since events cannot be split over multiple memory bins, there is a small gap at the end of each memory bin that cannot be utilized when an event does not fit completely in the remaining space. For smaller memory bins, in relation, this gap decreases the usable memory more drastically.

Consequentely, the number of events $\mathcal{E}$ that can be stored within the Hierarchical Memory Buffer depends on the total memory allocation for the buffer $\mathcal{S}_{buffer}$, the average size of an event $\mathcal{S}_{event}$, and the balance $\mathcal{B}$ of the event distribution (see Equation 4.2 on page 59):

$$\mathcal{E} = \frac{\mathcal{S}_{buffer}}{\mathcal{S}_{event}} \cdot \mathcal{B}$$

Using the definition in Equation 4.2 the worst case for the balance can be expressed with the number of hierarchy partitions p in relation to the number of memory bins b as $\mathcal{B} = 1 - \frac{p-1}{b}$; in a best case scenario the balance equals 1. Since the number of memory bins b results from the quotient of the total memory allocation for the buffer $\mathcal{S}_{buffer}$ and the size of the memory bins $\mathcal{S}_{bin}$, the number of events is:

$$\mathcal{E}_{best} = \frac{\mathcal{S}_{buffer}}{\mathcal{S}_{event}} \qquad \mathcal{E}_{worst} = \frac{\mathcal{S}_{buffer}}{\mathcal{S}_{event}} \cdot \left(1 - (p-1) \cdot \frac{\mathcal{S}_{bin}}{\mathcal{S}_{buffer}}\right)$$

[6] A hierarchy partition represents an element in the multi-dimensional memory buffer layout where, for instance, one dimension represents the call stack level and another dimension the event class.

However, this equation does not consider the gap at the end of each memory bin. The effectively usable buffer size $\mathcal{S}_{eff}$ due to this gap depends on the size and, therefore, the number of memory bins b and the size of the gap $\mathcal{S}_{gap}$:

$$\mathcal{S}_{eff} = b \cdot \left(\mathcal{S}_{bin} - \mathcal{S}_{gap}\right) = \frac{\mathcal{S}_{buffer}}{\mathcal{S}_{bin}} \cdot \left(\mathcal{S}_{bin} - \mathcal{S}_{gap}\right)$$

By using the effectively usable buffer size the number of events that can be stored within the Hierarchical Memory Buffer can be expressed as:

$$\begin{aligned} \mathcal{E}_{best} &= \frac{\frac{\mathcal{S}_{buffer}}{\mathcal{S}_{bin}} \cdot \left(\mathcal{S}_{bin} - \mathcal{S}_{gap}\right)}{\mathcal{S}_{event}} \\ \mathcal{E}_{worst} &= \frac{\frac{\mathcal{S}_{buffer}}{\mathcal{S}_{bin}} \cdot \left(\mathcal{S}_{bin} - \mathcal{S}_{gap}\right)}{\mathcal{S}_{event}} \cdot \left(1 - (p-1) \cdot \frac{\mathcal{S}_{bin}}{\mathcal{S}_{buffer}}\right) \end{aligned} \tag{5.1}$$

Figure 5.6 shows the estimated number of events based on Equation 5.1 modeled with the parameters $\mathcal{S}_{buffer} \in \{10\,\text{MiB}, 32\,\text{MiB}, 100\,\text{MiB}\}$, $\mathcal{S}_{event} = 3$, and $\mathcal{S}_{gap} = 32$ for different memory bin sizes $\mathcal{S}_{bin}$ and each a best case distribution and a worst case distribution for 20 and 80 hierarchy partitions. The increasing effect of the gap with decreasing memory bin size can be seen on the left end of each clustered histogram, where the number of events drops to about 50 %. The right end of each clustered histogram demonstrates the increasing effect of the event distribution balance with increasing memory bin size. For the best case scenario, that means a perfectly balanced distribution, the number of events hits its maximum at the size of one mebibyte because the effect of the gap is almost extinct. However, for the worst case scenarios the number of events drops drastically for high memory bin sizes. For a buffer size of 10 MiB there is no measuring point at 1 MiB because at this size only 10 partitions can be provided. For the same reason, the values for the worst case distribution with 80 partitions are incomplete. For larger memory buffer sizes the effect of the event distribution decreases because with an increasing memory buffer size the number of memory bins, which is most important for the event balance, increases as well.

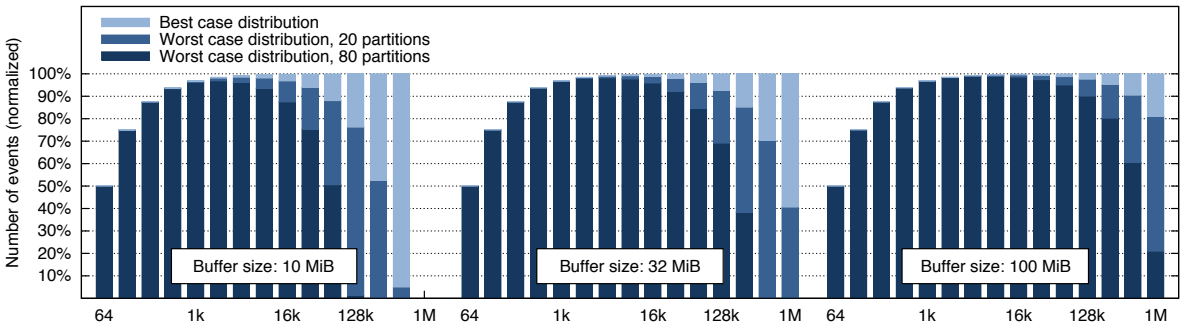


Figure 5.6: Number of events fitting within the memory buffer modeled for different memory bin sizes on the horizontal axis, different buffer sizes and different event distribution assumtions.

Next to the modeled behavior, the number of events that can be stored within the Hierarchical Memory Buffer was reviewed for the 21 applications and kernels. Figure 5.7 shows the minimum, maximum, and average number of events that could be stored within a 32 MiB memory buffer depending on the size of the memory bins (vertical axis). The number of events that could be stored varied strongly between the

applications, depending on the mixture of events, e.g., small events like region enter/exit to larger events like communication, and the size of the parameters of the events. Still, the two trends from the model are clearly visible again. For very small memory bin sizes the number of events decreases due to the increasing effect of the gap at the end of each memory bin. The relative decrease is more or less the same for all applications: the number of events at a memory bin size of 64 Bytes was between 51 % and 61 % of the maximum number of events for each application. Matters are quite different for larger memory bins sizes, where the event distribution mainly determines the utilization of the buffer. The strongest decline was recorded for *127.wrf2*, where the number of events dropped to 120,000 events or 1 % of the maximum. The *is* benchmark showed the weakest decline to 88 % of its maximum. These results are consistent with the event distribution that can be inferred from Figure 3.17. While *is*'s maximum calling depth is only four, *127.wrf2* has a maximum calling depth of 159 with most of its events clustered to a few of these call stack levels. In the majority of cases the maximum number of events was achieved with a memory bin size of four or eight kibibytes.

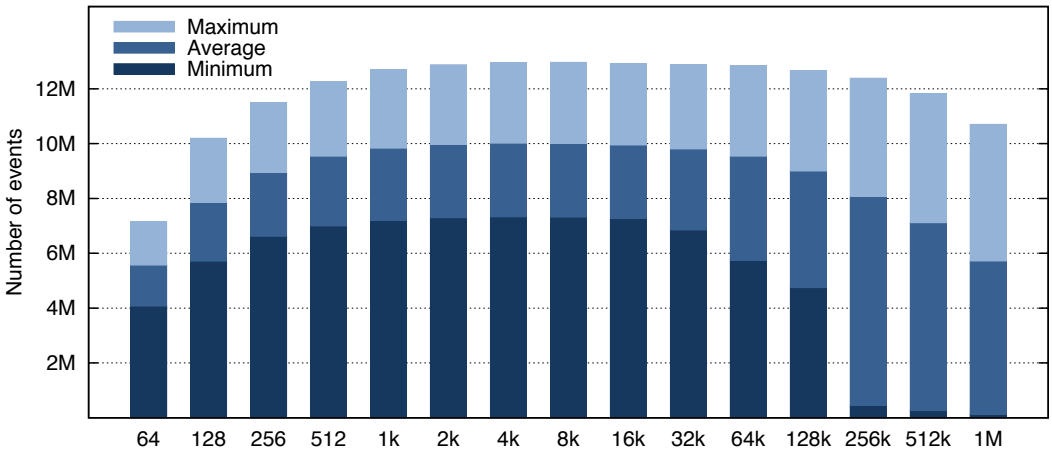


Figure 5.7: Maximum, minimum, and average number of events that could be stored within a 32 MiB memory buffer for different memory bin sizes.

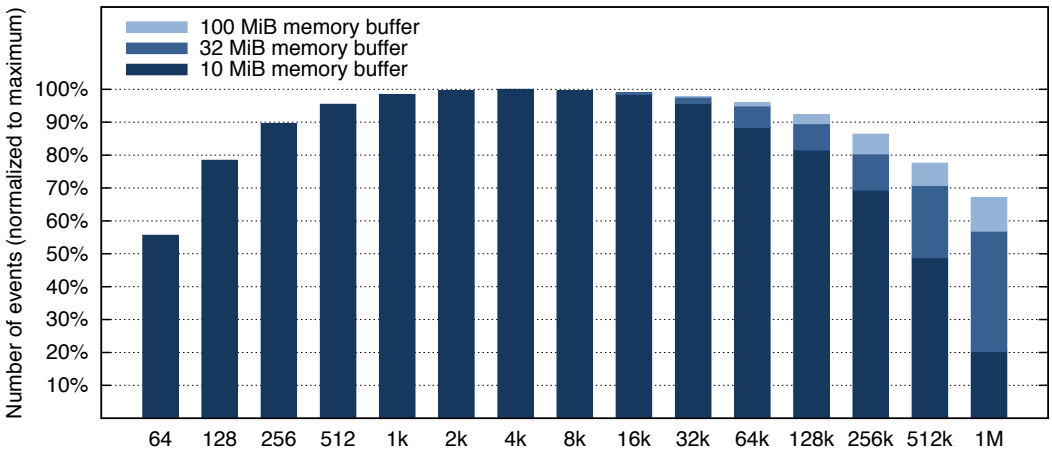


Figure 5.8: Average number of events that could be stored within a 10, 32, and 100 MiB memory buffer for different memory bin sizes, relative to each buffer sizes maximum.

Like in the modeled behavior, the effect of event distribution varies with the total size of the memory buffer. Figure 5.8 shows the average number of events that could be stored within a 10, 32, and 100 MiB memory buffer for different memory bin sizes. For a 10 MiB memory buffer there is a much sharper drop towards a memory bin size of 1 MiB than for the 100 MiB memory buffer. Also similar to the modeled behavior, the maximum number of events for most applications is achieved for a memory bin size of four and 16 kibibytes for a 10 MiB and 100 MiB memory buffer, respectively. Still, for *127.wrf2* the number of events that could be stored in a 100 MiB memory buffer with a memory bin size of 1 MiB is only 390,000, which is about 1 % of its maximum.
From the model and the figures it can be inferred that a feasible memory bin size in terms of memory utilization depends on the event distribution and the total memory buffer size. For each of the reviewed applications a memory bin size between 1 KiB and 16 KiB provides the best results.

Next to memory efficiency, the introduced overhead is decisive when choosing a feasible memory bin size. To measure the overhead of the OTFX prototype the time to fill a 32 MiB buffer was recorded and normalized to the time to write 1 million events. Again, the benchmark reviews the 21 applications and kernels. Figure 5.9 depicts the minimum, maximum, and average time to write 1 million events.

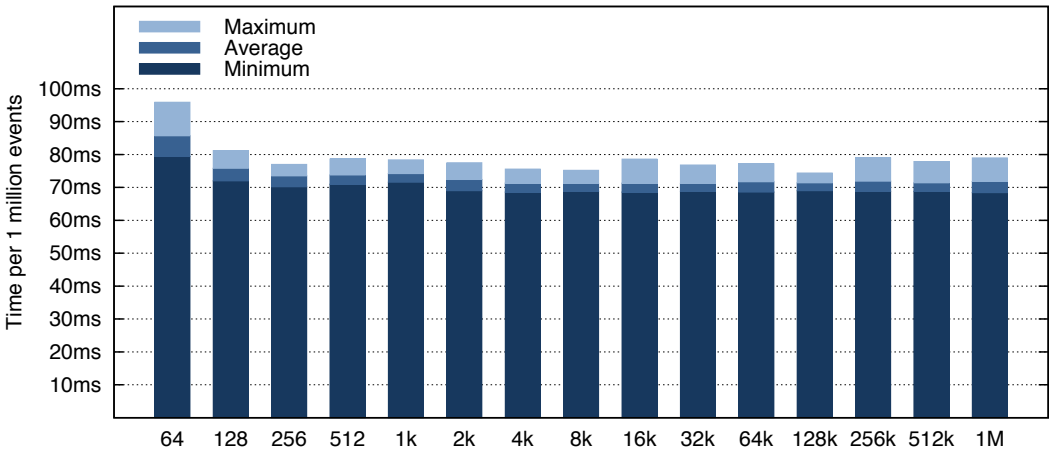


Figure 5.9: Maximum, minimum, and average runtime overhead for different memory bin sizes.

While the overhead depends on the mixture of events, e.g., smaller events can be stored faster than larger events, and varies between different applications, they all show practically the same behavior: the overhead is virtually constant for almost all memory bin sizes. Nevertheless, there is gradual rise in the overhead towards minimum and maximum memory bin size. For larger memory bins the overhead alike memory efficiency depends on the event distribution. In case of a disadvantageous event distribution memory efficiency decreases, which results in an increase of allocated memory per event and, therefore, an increase in time allocation time per event. This effect can be seen in the increase for the maximum overhead, while minimum and average overhead virtually stay the same. For smaller memory bins the overhead increases more steeply because here two effect accumulate. There is again a lower memory efficiency and, in addition, the overhead for allocation and management of the memory bins increases since the number of memory bins is inversely proportional to their size.
Nonetheless, this study shows that memory bin size scarcely influences the runtime overhead. For most of the reviewed applications a memory bin size of 128 Byte to 1 MiB delivers reasonable overhead.

5.3.2 Reduction of Hierarchy Partitions

These first measurements reviewed the performance of the Hierarchical Memory Buffer for traditional storing of events. However, the primary purpose of the Hierarchical Memory Buffer is to support different event reduction operations. Thereby, the goal is to efficiently reduce the number of events already stored in the memory buffer by removing hierarchy partitions from the memory buffer.

The following benchmark is designed to evaluate the overhead in case of an actual event reduction. It is implemented as a synthetic benchmark to be able to manage and exactly steer the grade of reduction. The synthetic benchmark writes 100 MiB of data, which is about 40 million enter and leave events, equally across 100 call stack levels. The writing pattern stores the first mebibyte of data to the deepest call stack level (i.e., 100), the second mebibyte on the second deepest call stack level (i.e., 99), and so on. By adjusting the total size of the Hierarchical Memory Buffer the grade of the reduction can be controlled. When writing 100 MiB of data to a buffer with a size of 100 MiB no reduction is triggered. For a buffer size of 99 MiB one hierarchy partition is discarded. Likewise, for a buffer size of 10 MiB the hierarchy partitions are stepwise reduced until only ten partitions remain. This way, each reduction operation reduces exactly 1 % of the stored events by removing one hierarchy partition, which contains 1 % of the distributed memory bins.

Figure 5.10 shows the time spent in the reduction operations within this benchmark for different memory bin sizes. The horizontal axis represents the size of the memory buffer in percent, i.e., the smaller the buffer size the higher the reduction. The time spent in the reduction operation and, thus, the overhead of the reduction increases linearly with the grade of reduction. The highest overhead is recorded for a memory buffer size of 64 bytes, which is 0.24 seconds or about 25 % for a reduction to ten percent of the original number of hierarchy partitions. For medium and large memory bin sizes, i.e., 512 bytes and above, the introduced overhead is at maximum between 1.2 and 4.6 percent.

While this benchmark uses the call stack based reduction strategy, it represent all types of reduction operations because all reduction operations are simply based on the removal of one or multiple hierarchy partitions. Therefore, from the results it can be concluded that a reduction operation generates overhead in the range of 1.4 to 5.1 % times the percentage of the revoked memory bins for a memory bin size greater than or equal to 512 bytes. Or in other words, the overhead is about 100 to 400 microseconds per mebibyte of revoked memory bins. For example, for an reduction of the event class 'performance metrics' containing one hardware performance counter value per enter/leave event, the number of memory bins that are revoked is about half of the total number (see also Figure 3.14) This means the overhead is in the range of 0.7 to 2.6 % of the total library time, in average 0.006 to 0.022 % of the total runtime of the reviewed applications and kernels, or 5 to 20 milliseconds for a 100 megabyte memory buffer. This underlines the statement from Section 3.4 that single event reduction steps should not be too large to avoid unnecessary information loss and bias.

In general, memory bin sizes of 1 KiB to 16 KiB provide best results in terms of memory efficiency, as well as writing and reduction overhead. With such sizes the Hierarchical Memory Buffer consists of enough memory bins to utilize the given memory well even for disadvantageous event distributions and provides enough memory bins to support applications with many hierarchy partitions. The event reduction operation with 100 to 400 microseconds per mebibyte of according memory bins creates a minor but noticeable interruption of the application. Still, this overhead is negligible small in comparison to interacting with a file system.

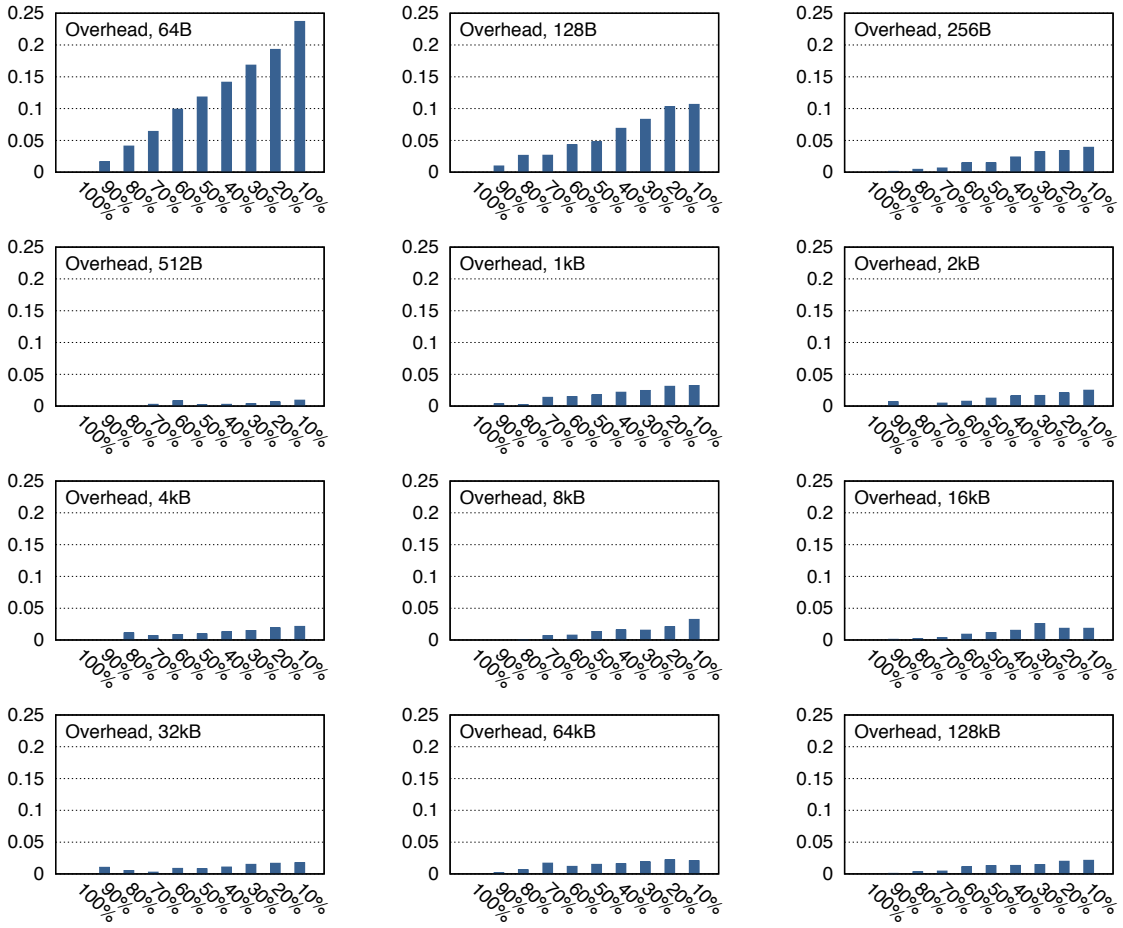


Figure 5.10: Overhead of event reduction operations (in seconds) for different memory bins sizes and different buffer sizes, i.e., grades of reduction.

5.3.3 Reduction by Duration

Next to reduction operations that reduce the number of hierarchy partitions, Section 3.4.4 introduces the reduction of code regions based on their duration. While the duration could also be used as a separate hierarchy dimension, in the OTFX prototype it is implemented as runtime filter and, therefore, all function calls shorter than the minimum duration are not stored, which drastically reduces the number of stored highly frequent function calls during runtime but keeps outliers that have an impact on the application behavior. This reduces the pressure on other reduction operations and the general overhead of storing heavily used functions. Thus, this technique could rather be classified as filter or selection technique but due its dependency on hierarchical data management it is so closely coupled with the Hierarchical Memory Buffer data structure that it is evaluated in this context.

Runtime Overhead

A first benchmark focuses on the runtime overhead that is introduced by tracking function duration and removing function calls shorter than a minimum duration. This synthetic benchmark writes 10 million events by looping over a pattern of ten nested function calls, i.e., ten enter events followed by ten leave events. All events are written with a fixed time interval of ten time units resulting in fixed durations for the nested function calls: ten units for the innermost function, 30 units for the second function, and up

to 190 units for the outer function. This pattern allows a step-wise reduction of recorded event trace data by increasing the minimum duration for function calls to be stored. The ten times nested function calls provide the possibility to reduce the amount of recorded data from 100 % (no function calls are filtered, minimum duration ≤ 10) to 0 % (all function calls are filtered, minimum duration > 190).
Figure 5.11 shows the results of this benchmark with different side effect compensation methods (see below). To compare the introduced overhead, a measurement of the OTFX prototype without the described modifications provides a baseline. A measurement of OTF2 serves as another reference. Both results are shown on the left in Figure 5.11.

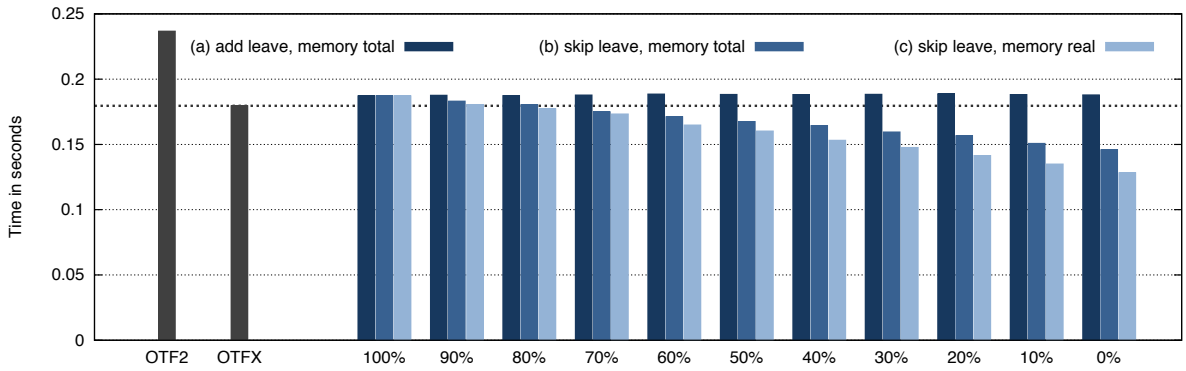


Figure 5.11: Runtime overhead code region reduction from 100 % (no function calls are filtered, minimum duration ≤ 10) to 0 % (all function calls are filtered, minimum duration > 190).

The runtime of the OTFX prototype with enabled code region reduction is represented by Measurement (c) in Figure 5.11. The runtime is linearly decreasing with an increasing minimum duration and, thus, a decreasing amount of stored event trace data. While this is the actual runtime of the OTFX prototype, obviously, the decreasing runtime is not caused by adding additional management to the Hierarchical Memory Buffer. There are two side effects that are introduced with the code region reduction method. First, the code region reduction leads to less stored event data, which results in less memory allocation for the Hierarchical Memory Buffer. Hence, the included time for memory allocation is reduced. Second, if a code region is not stored, the according enter event is first stored and later discarded but the leave event is not written at all. This leads to less write operations to the Hierarchical Memory Buffer with an increasing minimum duration. To provide a fair and correct comparison of the overhead of the OTFX prototype, there are two additional measurements that eliminate both effects. Measurement (a) in Figure 5.11 shows the runtime of the OTFX prototype with two modifications to eliminate both side effects by allocating the same amount of memory and always writing the leave event before an potential elimination. The runtime of Measurement (a) is constant (deviation < 1 %) regardless of the grade of function elimination and introduces an overhead of about 4 % compared to OTFX without the code region elimination. Measurement (b) keeps the constant memory allocation but like the original code region reduction implementation skips the writing of the leave event if the according function call is eliminated. Like in Measurement (c) the runtime in Measurement (b) is decreasing with an increasing minimum duration but with a slower decline.
While Measurement (a) proves that the OTFX prototype with code region reduction introduces no more than a constant overhead of about 4 %, Measurement (c) shows the real runtime of this approach, which is decreasing when more function calls are filtered since less events are stored and less memory is allocated.

Trace Size Reduction

Next to the introduced overhead, the capabilities to reduce the resulting event trace size during runtime are of highest interest. For that, eleven of the target applications and kernels are selected. The application kernels of the SPEC MPI 2007 benchmark suite are not included because the application traces did not contain any short running code regions due to the mature and optimized nature of the suite. Nonetheless, the remaining eleven applications and kernels, including the three real-life applications, provide a sufficient review to demonstrate the effects of the code region reduction.

For this evaluation a short-running code region is defined as a code region with a duration of less than 1 μs. With a minimum duration of 1 μs, all short-running code regions are eliminated while all important routines, including all communication routines, remain in the event trace. Table 5.4 demonstrates the capabilities of the code region reduction for the target applications and kernels. It shows the event trace sizes per location without and with duration filtering and the ratio to which the traces sizes are reduced.

Application	Trace Size Complete	Trace Size (min. duration 1 μs)	Ratio
BT	4 GB	4.3 MB	0.1 %
CG	406 MB	11.9 MB	2.9 %
EP	2.3 MB	0.2 MB	7.3 %
FT	23 MB	22.7 MB	99.4 %
IS	1.1 GB	2.0 MB	0.2 %
LU	134 MB	24.8 MB	18.3 %
MG	5.1 MB	0.9 MB	17.2 %
SP	142 MB	3.6 MB	2.6 %
Gromacs	2.5 GB	45.4 MB	1.7 %
Cosmo-Specs	2.3 GB	94.4 MB	4.0 %
Semtex	17 GB	9.3 GB	54.6 %

Table 5.4: Average event trace sizes per process with and without code region reduction.

For all applications and kernels that heavily use short-running code regions (see also Figure 3.18) the resulting trace sizes during runtime can be remarkably reduced. In particular, for *Gromacs*, *is*, and *bt* the trace sizes are reduced to 1.7 %, 0.2 %, and 0.1 %, respectively. Since *ft* includes almost no short-running functions, trace size is only hardly reduced. Table 5.4 also shows that for the reviewed applications and kernels, the reason that applications generate large trace sizes is usually the heavily use of short-running functions (*Gromacs*, *bt*, *is*). For all these applications the code region reduction provides an effective way to eliminate the short-running function calls and remarkably reduce the resulting event trace size.

Trace Analysis

To demonstrate the analysis on the reduced event traces, Figure 5.12 includes two screenshots of a visual analysis with Vampir [NAW+96]. While the upper screenshot shows the entire measurement, the bottom screenshot is zoomed in to a phase of about 3.8 ms. The fully recorded measurement can be seen on the upper half of each screenshot (white background); the measurement with code region reduction on the lower half (blue background). The timeline view with the events over time on the horizontal axis and

the locations on the vertical axis is shown on the left side. The function summary on the right depicts the total number of invocations for each code region. Both screenshots demonstrate that code region reduction does not alter the general application behavior; except for the missing short-running function calls. The function summary shows that the total number of function calls is reduced from about 4 billion to 68 million. The second screenshot additionally visualizes the process timeline of process zero in detail; with the calling depth on the vertical axis. The process timeline demonstrates that all short-running function calls on calling depth 10 and 11 are effectively eliminated while the outliers are kept; which is the key advantage over filtering based on the number of invocations.

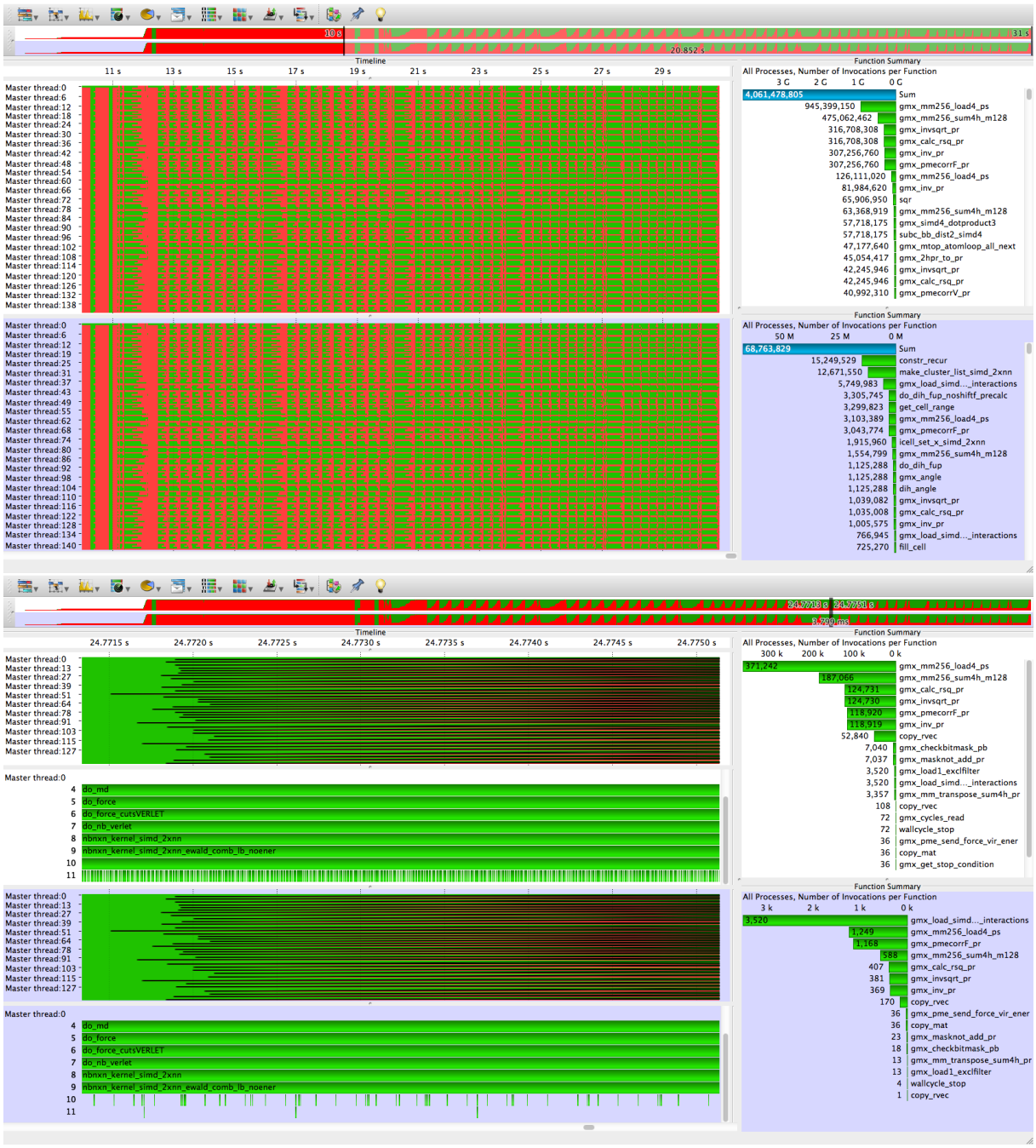


Figure 5.12: Event trace visualization with Vampir without and with code region reduction. Top: entire measurement; bottom: zoomed to an application phase of about 3.8 ms.

5.3.4 Analysis Techniques

As discussed in Section 4.5 all common analysis techniques for the Hierarchical Memory Buffer like timeline visualization, statistical summaries, or message matching rely on a basic forward traversal, i.e., reading all events. The complexity of such a basic forward traversal is $\mathcal{O}(p + n \log p)$ with n being the number of events and p the number of non-empty hierarchy partitions. In addition, the basic forward traversal can be restricted to a subset of hierarchy partitions, for example, to those partitions that contain communication events for a communication analysis. In this case, the complexity is reduced to $\mathcal{O}(p_h + n_h \log p_h)$ with p_h the number of according hierarchy partitions and n_h the number of events within these hierarchy partitions (see Section 4.5). At any rate, the complexity for a basic forward traversal is $\mathcal{O}(p + n \log p)$ instead of $\mathcal{O}(n)$ for a flat continuous memory buffer since all events stored in the different hierarchy partitions must be merged in a way that they are handed to the user ordered by their time stamp. However, such a complexity can only be achieved by using a heap data structure (or similar) for an efficient merging of events. Therefore, reading performance for the Hierarchical Memory Buffer mainly depends on the number of non-empty hierarchy partitions.

Figure 5.13 sets the results of a stress test in comparison to the reading performance for the target applications and kernels. The stress test is designed to simulate worst case behavior. It equally distributes n = 1 million events on the given number of hierarchy partitions, in this case call stack levels, by looping over the simple pattern of p enter events followed by p leave events, with p being the maximum number of hierarchy partitions. In comparison the target applications and kernels are evaluated by reading the first million enter/leave events.

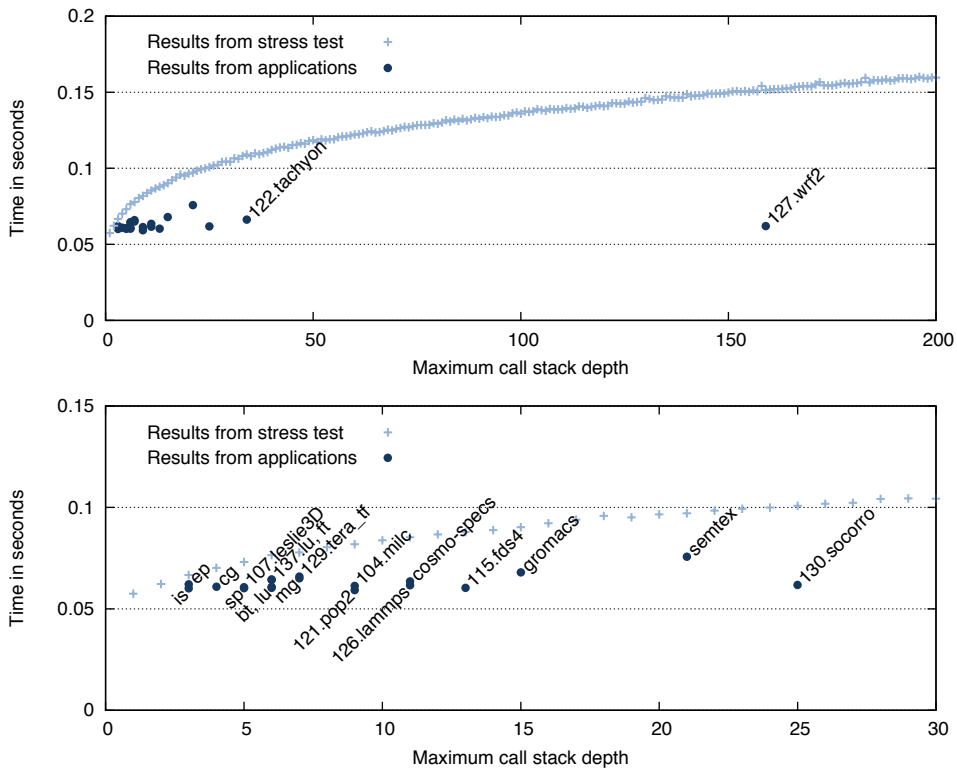


Figure 5.13: Time to read events depending on the maximum calling depth for a stress test and for the real-life applications and kernels. Top: Scaled to maximum calling depth of 200. Bottom: Zoomed to a maximum calling depth of 30.

The stress test shows that in worst-case the merging can be done in logarithmic time complexity as indicated by the complexity of the forward traversal $\mathcal{O}(p + n \log p)$. However, the evaluated applications and kernels reflect a different behavior than the stress test. Most events are recorded on only one or a few call stack levels, which leads to a much lower cost for the *shift down* operation on the heap data structure, namely $\mathcal{O}(p_f)$, instead of $\mathcal{O}(p)$, with $p_f < p$ being the number of the few hierarchy partitions most events are stored in. Therefore, the reading performance for the target applications and kernels depends not only on the total number of hierarchy partitions but on the distribution of events among these hierarchy partitions, as well. The target applications and kernels show drastically higher reading performance than the stress test caused by a nearly constant time complexity for the shift down operation.

5.3.5 Message Matching on Incomplete Communication Data

To allow a correct message matching and, thus, a correct communication analysis on incomplete communication data, Section 4.6 introduced the sequential message identifier. First and foremost, it is important to investigate if this approach can actually be used to reduce the amount of communication data with event reduction. In particular, only if message envelopes are used multiple times, communication data can be reduced. Otherwise, if each message envelope is used only once, the amount of stored data would be doubled: the data for each communication event is stored in the memory buffer and also in parts in the internal management data structure (message envelope and sequential message identifier). In that case, even if all message events in the trace are deleted, the data is still kept in the internal management data structure. Thus, the key indicator for the overall feasibility and the memory saving potential is the number of messages per message channel. It can be assumed, naturally, that message envelopes are hardly ever reused, since the idea of the message envelope, especially the message tag, is to contain distinguishing information. However, it can also be considered that for instance iterative application reuse message envelopes frequently by using the same message tags in each iteration.

To determine the range of the number of messages per message envelope a statistical survey evaluates different applications: the molecular dynamics package Gromacs [HKS08] and the NAS Parallel Benchmarks [BLBS92]. The NAS Parallel Benchmarks are evaluated at a scale of 4096 processes and the problem size E because it can be assumed that the number of messages per message envelope decreases with an increasing number of locations, . In comparison, Gromacs was surveyed at different scales to show the effects of an increasing number of participating locations.

Table 5.5 shows the number of messages per message envelope for the survey applications and in comparison the total number of messages per location. The applications EP and FT of the NAS Parallel Benchmarks are not included since they only use collective communication operations. The application IS is also not included because it does not support problem size E; on other problem sizes it uses exactly one point-to-point communication per location. Due to the enormous generated trace data, the Gromacs application was surveyed for 10.000 iterations while production runs can run for up to one million iterations. Thus, it can be assumed that the number of messages per message envelope for an production run is about 100 times higher than depicted here. Gromacs shows the expected trend of a decreasing number of message channels for an increasing number of locations. From Table 5.5 it can be inferred that all surveyed applications reuse the message envelopes in a sufficient way. Therefore, the survey rejects the previous assumption that message envelopes are hardly reused. In fact, message envelopes are regularly reused, which makes the described approach feasible.

Application		Messages per Envelope			Messages per Location		
		Minimum	**Average**	Maximum	Minimum	**Average**	Maximum
Gromacs	P = 384	1	**14171**	64933	509492	**641395**	818009
Gromacs	P = 768	20	**7016**	50038	418765	**444361**	464050
Gromacs	P = 1056	10	**3827**	50028	460740	**517933**	550266
NPB BT	P = 4096	251	**251**	252	192780	**192780**	192780
NPB CG	P = 4096	2626	**7214**	7979	101000	**101000**	101000
NPB LU	P = 4096	306722	**306723**	306724	1226892	**2415440**	2453780
NPB MG	P = 4096	51	**894**	1279	15042	**15532**	20550
NPB SP	P = 4096	502	**16033**	31563	384780	**384780**	384780

Table 5.5: Number of messages per message envelope and per location.

At runtime, for each message envelope an internal management data structure keeps an 16 byte entry (based on OTF2): 4 bytes each for the communication partner, message tag, communicator and sequential message identifier (see also Listing 4.2). The data structure can be organized as associative array that grants access in logarithmic time complexity. The number of entries in the management data structure depends on the number of different message envelopes, which correlates with the number of communication partners within a given communicator and is bounded by the number of processes p, the number of different message tags t, and the number of different communicators c. Hence, the memory requirement in the management data structure is in $\mathcal{O}(p \times t \times c)$ and the access time in $\mathcal{O}(log(p \times t \times c))$.

Application		Message Envelopes per Location		
		Minimum	**Average**	Maximum
Gromacs	P = 384	22	**45.3**	130
Gromacs	P = 768	17	**63.3**	153
Gromacs	P = 1056	18	**135.3**	369
NPB BT	P = 4096	768	**768.0**	768
NPB CG	P = 4096	14	**14.0**	14
NPB LU	P = 4096	4	**7.9**	8
NPB MG	P = 4096	12	**17.4**	48
NPB SP	P = 4096	24	**24.0**	24

Table 5.6: Number of different message envelopes per location.

Therefore, the memory allocation, as well as the access time, benefit from a small number of message envelopes, while a large number of different message envelopes may overwhelm the management data structure in terms of memory allocation and overhead. Table 5.6 contains the number of different message envelopes per location. For all surveyed applications the number of message envelopes is very small and far below a theoretical maximum $p \times t \times c$. Although there might exist applications with very excessive communication patterns, in most applications processes communicate mainly with their neighboring processes, with only a few large messages instead of many small messages, and within very few different communicators. Hence, the values for p, t, and c are each small, keeping the number of

message envelopes small, too. From Table 5.6 it can be concluded that memory allocation as well as access time for the internal management data structure are small.
Finally, Table 5.7 shows the memory overhead in more detail. First, it can be seen that the memory used for the internal management data structure (column Message ID) is negligible for all surveyed applications. Second, the additional event trace data – caused by the additional parameter keeping the sequential message identifier – directly depends on the total number of MPI point-to-point messages (see also Table 5.5). The memory overhead of about 2.6 to 10.2 % is justifiable, in particular, since the share of MPI point-to-point events within the complete event trace data is typically in the single-digit percentage range or less (see also Figure 3.14).

Application		Additional memory requirements		
		Message ID	Trace Data	Ratio
Gromacs	P = 384	0.7 KiB	1871 KiB	8.3 %
Gromacs	P = 768	1.0 KiB	1294 KiB	9.0 %
Gromacs	P = 1056	2.2 KiB	1505 KiB	9.3 %
NPB BT	P = 4096	12.3 KiB	377 KiB	2.6 %
NPB CG	P = 4096	0.2 KiB	292 KiB	5.5 %
NPB LU	P = 4096	0.1 KiB	8929 KiB	10.2 %
NPB MG	P = 4096	0.3 KiB	42 KiB	5.0 %
NPB SP	P = 4096	0.4 KiB	1121 KiB	4.2 %

Table 5.7: Additional memory requirements per location.

5.4 Case Study: The Molecular Dynamics Package Gromacs

This section evaluates the contributions of this thesis on the basis of the molecular dynamics package Gromacs [HKS08]. It demonstrates how the contributions, in particular, can improve the event-based performance analysis of long-running applications that exceed the capabilities of traditional event tracing approaches for both, the introduced bias, as well as the resulting event trace sizes. The first part of this section focuses on the impact of intermediate buffer flushes on the recorded application behavior. The second part evaluates how the contributions of this thesis can be applied to drastically increase memory efficiency, on the one hand; and on the other hand, use event reduction techniques to reduce the number of stored events during runtime and, therefore, avoid intermediate buffer flushes.

5.4.1 The Molecular Dynamics Package Gromacs

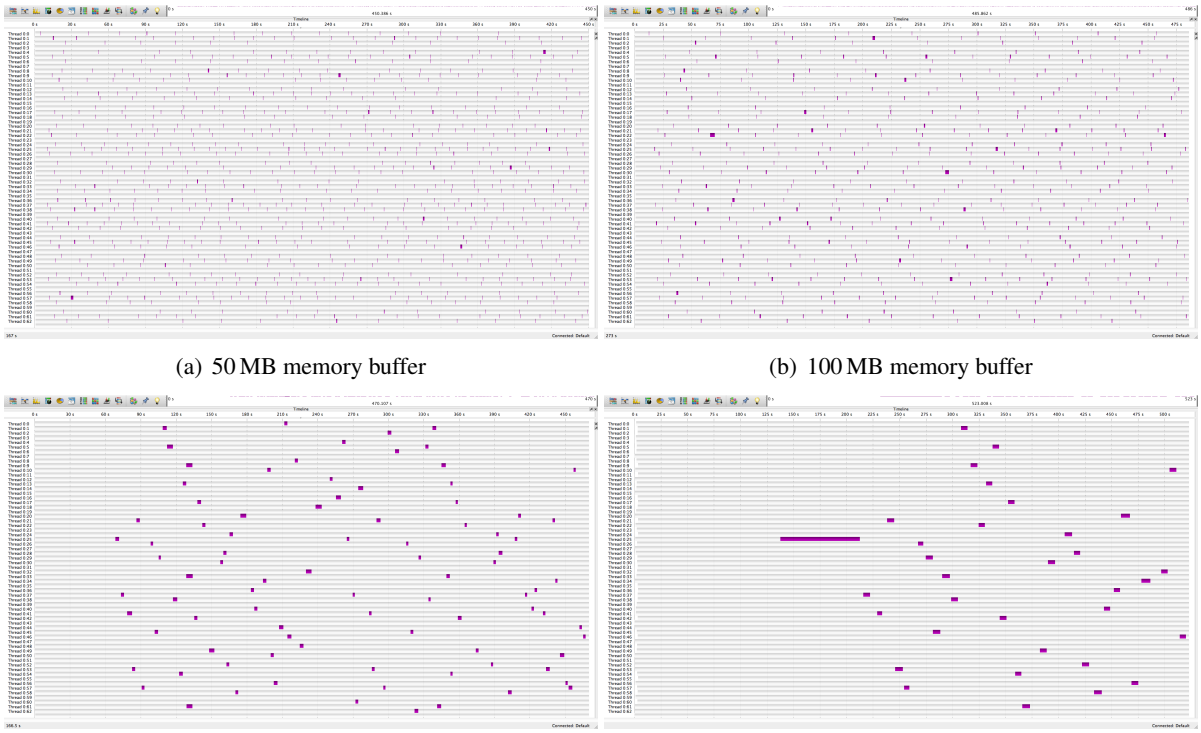
Gromacs is a versatile package to perform molecular dynamics, i.e., simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions. Because Gromacs is extremely fast at calculating non-bonded interactions that usually dominate simulations, many groups are also using it for research on non-biological systems such as polymers. It is one of the fastest and most popular software packages available for molecular dynamics. Since its origins in

1991 the package grow to about 1.8 million lines of code with an estimated effort of over 6000 person months [GRO14]. Within the EU project CRESTA [CRE14] development is supported by an in-depth performance analysis with Score-P and Vampir to optimize performance for exa-scale HPC systems.
However, tracing an application run of Gromacs produces enormous amounts of event data. An exemplary simulation of 50 iterations with 144 processes already results in an average trace size of about 703 MiB per location using Score-P and OTF2. Running the sample simulation with 10.000 iterations would lead to an estimated average trace size of about 125 GiB per process[7]. It is obvious that actual production runs that use up to one million iterations could not be recorded with the full detail provided by traditional event tracing approaches. While the resulting trace size is an urgent challenge to solve, the bias caused by intermediate buffer flushes can render a measurement run completely useless and, therefore, may prevent a successful performance analysis.

5.4.2 The Bias Caused by Intermediate Buffer Flushes

As stated above, the recording of 50 iterations already results in an average trace size of about 703 MiB. Thereby, the event data is unequally distributed over the resulting 144 event trace files. Gromacs uses, next to a domain composition, also a function decomposition that splits the locations in two groups with a ratio that depends on the number of active locations. In this case the ratio is 3:1, i.e., three out of four locations compute the particle-particle interaction (PP) while one other computes the Particle Mesh Ewald method (PME). As a result, the trace sizes for the different locations are divided in two groups, too. The trace sizes of the first group range from 641 MiB to 1.4 GiB and the trace size of the second group ranges from 33 to 38 MiB.
Whenever an internal memory buffer is exhausted, its data is flushed to a file. This causes a noticeable interruption of the according locations since it is stalled until the file interaction completes. Due to the deviations in Gromacs, all locations flush their memory buffer at a different time. Figure 5.14 shows the distribution of buffer flushes during runtime for different memory buffer sizes. With an increasing memory buffer size, the number of buffer flushes decreases but their duration increases. While for the 50 MiB memory buffer there occur a total of 1985 buffer flushes with an average interrupt of 0.13 seconds, for the 1 GiB buffer there appear 34 buffer flushes with an average interrupt of 8.4 seconds.
Regardless of the memory buffer size the pattern of the memory buffer flushes is highly irregular. Whenever those memory buffer flush occur during or near an inter-process dependency, such as a communication operation, the interrupt causes a modification of the original application behavior. Figure 5.15 demonstrates this effect for a memory buffer size of 1 GiB since the effect is most easily visible there. Within the application there are several global communication operations. Up to the first buffer flush of process 25 at about 140 seconds these communication operations (red) require only little application time as can be seen by the share of red within white, which contains everything else within the application. However, during the first buffer flush all other processes wait for process 25 to finish its buffer flush and engage in the global communication operation. After that, each buffer flush causes the same behavior, which can be seen by the red blocks on each process that have approximately the same length as the buffer flush of one of the processes. Comparing the application behavior before and after the first buffer flush clearly demonstrates the effect of uncoordinated intermediate memory buffer flushes on the recorded application behavior.

[7] Values are estimated with the scorep-score tool based on a profile run [KRM+12].



(a) 50 MB memory buffer

(b) 100 MB memory buffer

(c) 500 MB memory buffer

(d) 1 GB memory buffer

Figure 5.14: Vampir screenshot displaying the distribution of buffer flushes (violet).

Figure 5.15: Vampir screenshot displaying the distribution of buffer flushes (violet) for a 1 GB buffer and the resulting MPI wait times (red).

Figure 5.16 supports this conclusion from another point of view. It displays the runtime of Gromacs for different memory buffer sizes. The blue share of each bar depicts the runtime of Gromacs excluding memory buffer flushes and their influence. The light violet share equals the time actually spent in memory buffer flushes while the dark violet share represents the time spent waiting for other locations to finish their buffer flushes. Figure 5.16 shows that, first, the sum of time spent in memory buffer flushes is fairly constant regardless of the buffer size. Second, the overhead caused by locations waiting for other processes to finish their memory buffer flushes is slightly increasing with an increasing memory buffer size (255 to 285 seconds). Third and most important, the overhead caused by memory buffer flushes is about 64 times higher than the time actually spent in buffer flushes and is about 55 % of the total runtime.

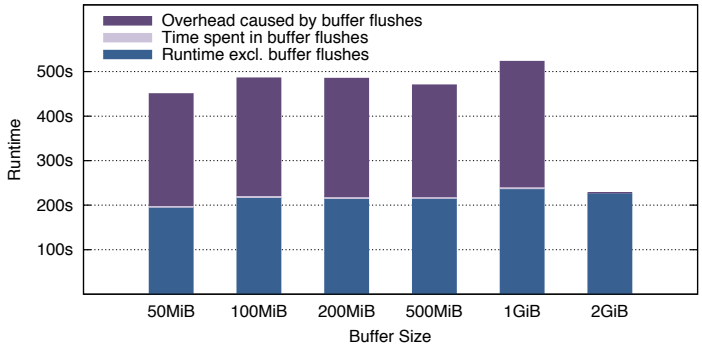


Figure 5.16: Overhead caused by uncoordinated intermediate buffer flushes.

Both, Figure 5.15 and 5.16 demonstrate the disruptive effect of uncoordinated memory buffer flushes. In particular, Figure 5.15 highlights that all of the recorded behavior beyond the first initiated buffer flush must be considered incorrect. For typical buffer sizes of 50 or 100 MiB this means that 99 % or 97 %, respectively, of the entire application runtime must be considered incorrect. Therefore, the contributions of this thesis to avoid intermediate buffer flushes provide a major improvement to accurately record the behavior of long running applications.

5.4.3 In-memory Event Tracing for Long Application Runs

Tracing long application runs always requires adaptions to the recording procedure; otherwise the resulting data volumes bias the recorded application behavior, overstrain file system capacities and massively slow down or even preclude subsequent analysis steps. This section evaluates to which extend the contributions of this thesis can help not only to reduce the trace data size but to keep data within a single fixed-size memory buffer to enable in-memory event tracing even for long-running applications.

As stated above, a trace-based analysis of the molecular dynamics package Gromacs is limited to only a few of the up to 1 million iterations for real-life production runs. Recording only the first 50 iterations already results in an average of about 700 MiB of data per location with a maximum amount of 1.4 GiB per location. In the following, a memory buffer size of 100 MiB is taken as basis for the measurements and estimations. Thus, given a memory buffer of a 100 MiB, a measurement using OTF2 could only record the first three iterations until the first memory buffer is exhausted and, therefore, must be flushed to the file system. By using OTFX, the prototype that includes the enhanced encoding techniques and the Hierarchical Memory Buffer, the trace size is reduced to 400 MiB for the largest trace file. Still, this would only allow the recording of the first 12 iterations within a 100 MiB memory buffer.

To further reduced the trace size, the event reduction strategies must be enabled. Figure 5.17 shows that there is an enormous potential for the elimination of short-running function calls and a sufficient distribution of events across the different call stack levels. While the call stack level reduction is usually triggered automatically whenever the memory buffer is exhausted, for this study it is controlled manually to reduce the stored calling depth to 7 and 6, which leads to a storage of about 40 % and 4 %, respectively.

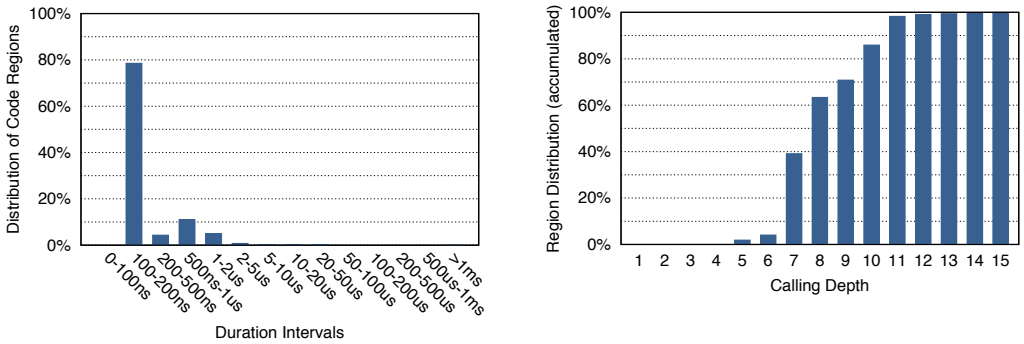


Figure 5.17: Distribution of code regions by time (left) and calling depth (right).

Table 5.8 presents the trace size reduction and the resulting estimated iterations that could be recorded within a 100 MiB memory buffer. With enabled code region reduction that filters all function calls with a duration shorter than 1 μs the trace size is remarkably reduced to about 1.9 % of its original size. By adding the calling depth reduction the trace size could be further reduced to 1.4 % and 0.1 % for a reduction to a calling depth of 7 and 6, respectively.

	Maximum data size	Percentage	Iterations
OTF2	1433 MiB	100 %	3
OTFX	398 MiB	27.8 %	12
Code region reduction	27.7 MiB	1.9 %	180
Reduction to calling depth 7	19.4 MiB	1.4 %	257
Reduction to calling depth 6	0.94 MiB	0.1 %	5321

Table 5.8: Trace sizes for different reduction strategies.

While the number of recordable iterations is remarkably increased from three iterations to over 5000 with enabled code region and calling depth reduction, it is still far from the target of 1 million iterations. Although this already demonstrates the potential of the enhanced encoding techniques and the event reduction strategies, standing by themselves it is not enough to support long-running applications. A possibility to decrease the initial number of recorded events is a selection of only a few iterations; either statically, e.g., record every 200th iteration or dynamically, e.g., record only representatives of iteration classes as discusses in Section 3.2. An alternative is to exclude all code regions that would usually by inlined by the compiler from being recorded at all. With the latter approach another measurement was recorded with 10,000 iterations, which results in a trace size of 667 MiB at maximum.

Table 5.9 shows again the trace size reduction and the resulting estimated iterations that could be recorded within a 100 MiB memory buffer. With enabled code region reduction the trace size is still remarkably reduced to 3.2 % of its original size in spite of the fact that all inline functions are already excluded.

Adding calling depth reduction decreases the trace size further to 2.1 % and 1.2 % for a reduction to a calling depth of 7 and 6, respectively. This allows the recording of about 120 thousand iterations, which is still one order of magnitude below the target goal. By also enabling the event class reduction that discards events from all event classes other than enter/leave events and an additional reduction to a maximum calling depth of 5, the trace size could be reduced to 0.1 %. This way, it becomes possible to reach the target and record over 1 million iterations.

	Maximum data size	Percentage	Iterations
OTF2	667 MB	100 %	1500
OTFX	196 MB	29.4 %	5104
Code region reduction	21.2 MB	3.2 %	47,169
Reduction to calling depth 7	14.2 MB	2.1 %	70,422
Reduction to calling depth 6	8.25 MB	1.2 %	121,212
+ reduction of event classes	2.48 MB	0.4 %	403,225
Reduction to calling depth 5	0.97 MB	0.1 %	1,030,927

Table 5.9: Trace sizes for different reduction strategies with function inlining.

However, this tremendous trace size reduction of almost three orders of magnitude comes with a reduction of events stored within the trace (53 million to 214 thousand). This immediately raises the question about the usefulness of the remaining events or the usability of the event reduction in general. It is obvious, that the reduced trace cannot deliver the same level of detail as a complete trace; a reduced trace is far more coarse. But can the remaining coarse trace still contribute to the two main targets of performance analysis: better understand the application behavior and identify performance issues? Figures 5.18 and 5.19 present a screenshot of a visual performance analysis of Gromacs in the 10,000 iteration version without and with reduction to calling depth five. The first figure shows an overview of the entire application and the second includes a small section of approximately three iterations. Both show a timeline view of the first twelve processes with the application behavior over time on the horizontal axis and the processes on the vertical axis (top, left), a detailed call stack of process zero (bottom, left) and a function summary listing the inclusive function time (right). The figures visualize the prominent functions on calling depth five *do_force* and *gmx_pme_do* in yellow[8] and blue, respectively, the rest of the application functions in green and MPI communication in red.

What can be seen first hand in Figure 5.18 are three things: First, the remaining events of the reduced version equal exactly the complete version, which is highlighted by the timeline view, as well as the function summary. Second, any analysis option related to MPI communication is impossible, due to the reduction of all event classes except enter/leave. Third, all functions with a calling depth higher than five are not contained in the reduced trace resulting in a considerably lower detail, which can be seen in the timeline view, as well as the process timeline of process zero.

Figure 5.19 unveils the lack of detail within each iteration, in particular, the missing communication. However, the reduced trace clearly identifies the overall program behavior. It illustrates the function decomposition within each group of four processes and the iterative blocks of the application. These two characteristics are even highlighted more clearly because of the reduced number of function calls.

[8]For better visibility most nested functions of *do_force* (except MPI communication) are marked yellow, too.

Due to the maturity of Gromacs it does not contain any obvious performance issues. Nonetheless, performance issues such as load imbalances that would manifest within the five remaining call stack levels would still be visible. Inefficient communication patters, however, would not be visible.

Figure 5.18: Event trace visualization with Vampir without (top, white background) and with reduction to calling depth five (bottom, blue background).

Figure 5.19: Event trace visualization with Vampir without (top, white background) and with reduction to calling depth five (bottom, blue background) zoomed to approximately three iterations.

Applying an automatic analysis with Scalasca delivers similar results. Figures 5.20 and 5.21 depict a screenshot of Cube with the results of an automatic analysis of Gromacs in the 10,000 iteration version without and with reduction. While, again, all functions with a calling depth higher than five and all communication events are missing in the reduced trace, the remaining events of the reduced version equal exactly the unreduced version. The dominating functions in both groups of the function decomposition *do_force* and *gmx_pme_do* are highlighted, as well as their runtime distribution in the system tree view.

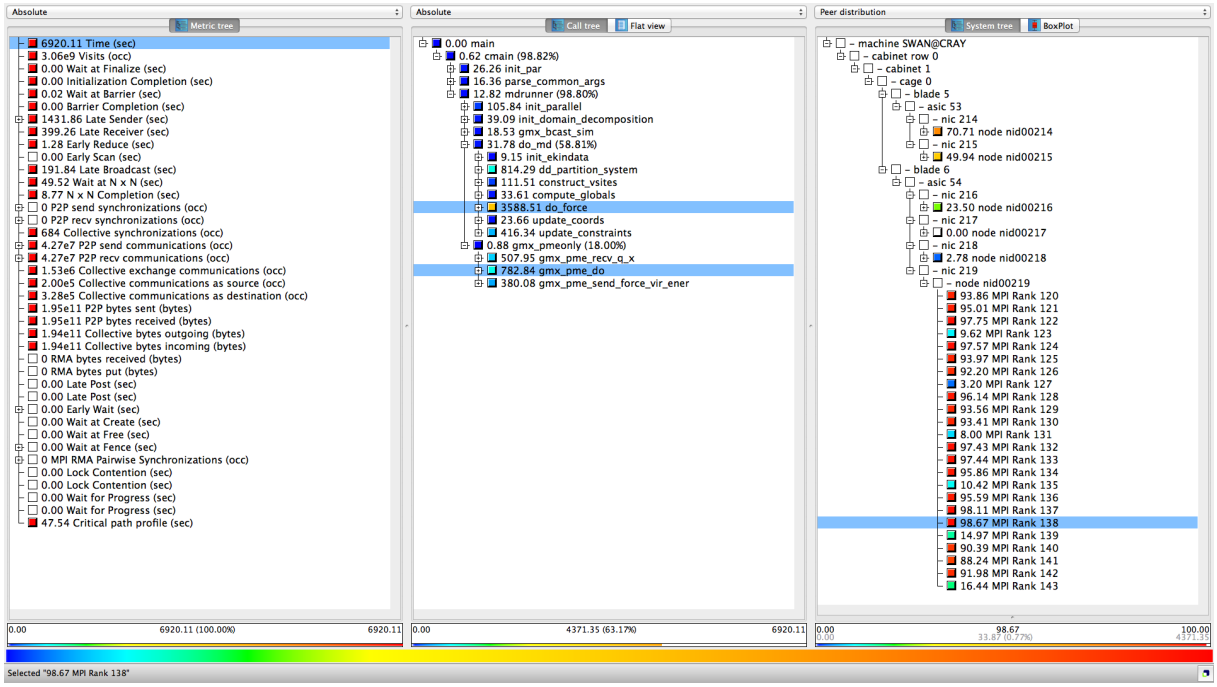


Figure 5.20: Automatic analysis results of Scalasca in Cube without reduction.

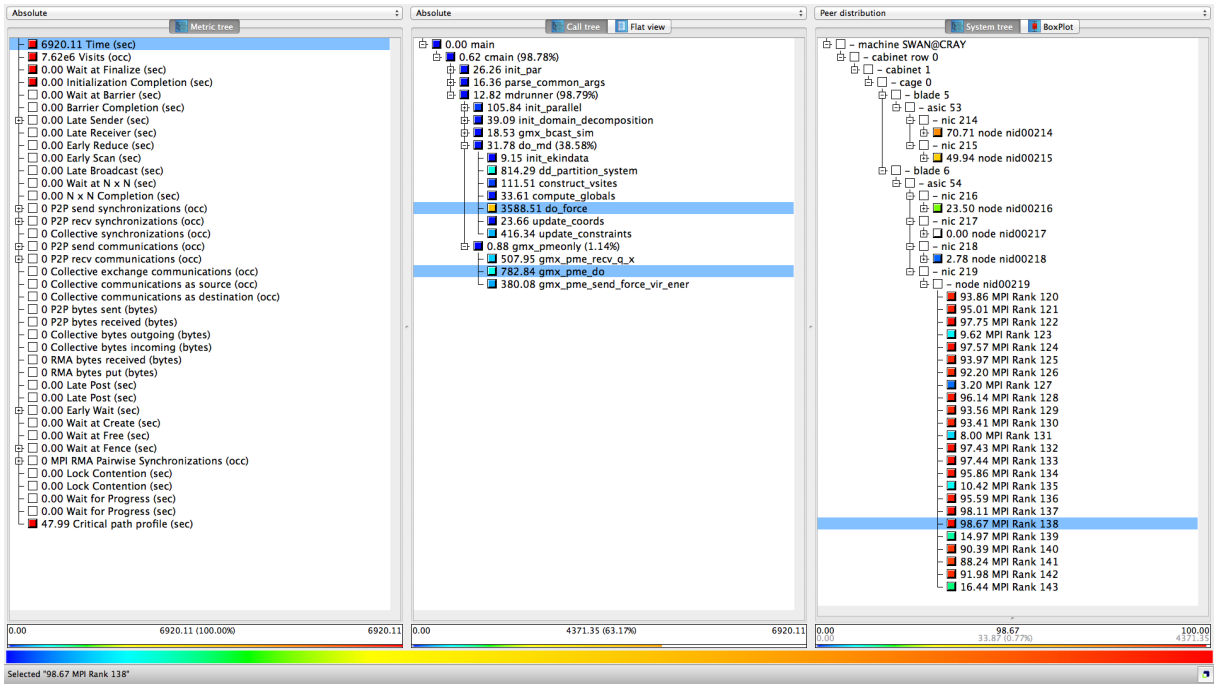


Figure 5.21: Automatic analysis results of Scalasca in Cube with reduction to calling depth five.

This case study demonstrates, first, that uncoordinated intermediate memory buffer flushes can lead to a drastic bias on the application leading to potential incorrect behavior after the first memory buffer flush occurs. For long-running applications this bias renders most of the recorded application useless. Second, the combined concepts including the enhanced encoding techniques, event reduction, and a new filter based on function duration within the Hierarchical Memory Buffer remarkably reduce the resulting trace up to three orders of magnitude and keep an entire measurement within a single fixed-size memory buffer, while still providing a coarse but meaningful analysis of the application. Hence, they provide an essential improvement to the event-based performance analysis, in particular, for long-running applications.

5.5 Summary

This chapter presents an evaluation of the enhanced encoding techniques and the Hierarchical Memory Buffer data structure. The first part of this chapter focuses on the enhanced encoding techniques and shows the effects of each individual encoding technique as well as a comparison to other well-established event trace formats. These encoding enhancements proof to be effective and reduce memory allocation during runtime by a factor of 3.3 to 7.2 for the evaluated applications and application kernels while in the same do not introduce any additional overhead on the tracing library. In fact, the overhead is slightly reduced since less memory must be allocated. With these results the enhanced encoding techniques outperform the memory efficiency of well-established general purpose compression libraries, which introduce a five times higher overhead. In comparison to other event trace formats, the OTFX prototype that contains all presented encoding techniques consumes about 91 % less memory than VampirTrace, about 79 % less memory than OTF, about 71 % less memory than Epilog and about 70 % less memory than OTF2. Therefore, the enhanced encoding techniques provide a remarkable improvement to existing event trace formats.

The second part studies the Hierarchical Memory Buffer data structure, which uses a hierarchical layout and a highly dynamic distribution of memory bins instead of a single flat continuous memory segment like other event trace libraries. This novel design is specifically tailored to efficiently support event reduction during runtime. For this purpose a size of 1 to 16 KiB for the individual memory bins provides best results in terms of memory efficiency, as well as writing and event reduction overhead. With these sizes the Hierarchical Memory Buffer consists of enough memory bins to utilize the given memory well even for disadvantageous event distributions and provides enough memory bins to support applications with many hierarchy partitions.

A study of the real-life application Gromacs uncoveres the drastic bias introduced by uncoordinated intermediate buffer flushes and evaluates the capabilities of the event reduction techniques to avoid buffer flushes by gradually reducing the amount of events stored in the Hierarchical Memory Buffer. The study demonstrates that the enhanced encoding techniques combined with the event reduction presented in this thesis can remarkably reduce the resulting trace size up to three orders of magnitude, while still providing a coarse but meaningful analysis of the application. The combination of the new filtering by code region duration, the enhanced encoding techniques, and event reduction allows an in-memory event tracing workflow even for large real-life applications, such as Gromacs. Hence, they provide an essential improvement to event-based performance analysis, in particular, for long-running applications.

6 Conclusion and Outlook

This chapter summarizes the contributions of the thesis and draws a conclusion. Furthermore, an outlook on future work and the extension of the in-memory workflow to an entire online event tracing workflow complete this thesis.

Summary and Conclusion

This thesis emphasizes the benefits of an in-memory workflow for event-based performance monitoring and presented strategies to realize such a workflow.

After an introduction into the field of performance analysis and an overview of well-established performance analysis tools and their approaches, the motivation for this thesis is formulated by three essential challenges, yet unsolved in event-based performance analysis. First, the limits of parallel file systems in the number of resulting trace files, second, the enormous data volumes generated by event tracing, and, third, the measurement bias introduced by uncoordinated intermediate memory buffer flushes.

The first central part of this thesis presents three key steps to enable an in-memory event tracing workflow: selection and filtering, encoding and compression, and event reduction. It introduces new enhanced encoding techniques and the novel approach of event reduction that dynamically adapts trace size during runtime to the given memory allocation. The combination of both allows to keep the data of an entire measurement within a single fixed-size memory buffer, which is the premise for an in-memory event tracing workflow. Such an in-memory event tracing workflow meets all three motivating challenges by not only overcoming the limitations of current parallel file systems but eliminating the overhead of file system interaction altogether. In addition, the new enhanced encoding techniques and the novel event reduction result in remarkably smaller trace sizes. In particular, the runtime event reduction is capable to keep the data within a memory buffer of any given size. Furthermore, the in-memory workflow completely avoids intermediate memory buffer flushes and, therefore, minimizes measurement bias and allows event tracing of long running applications, unfeasible with previous approaches.

The second central part introduces the Hierarchical Memory Buffer, a new event data representation that uses a hierarchical layout and a highly dynamic distribution of memory bins instead of a single flat continuous memory segment like current event tracing libraries. It allows to perform the aforementioned event reduction operations with minimal overhead. Furthermore, such a hierarchy-based event representation supports new event selection and filter operations unfeasible with a traditional flat continuos memory buffer layout. Such a new filter method is a filtering based on the duration of code regions, which effectively eliminates all short-running functions while keeping outliers important for performance analysis. In addition, several common analysis requests, such as summaries or communication analysis, can benefit from a hierarchy-aided traversal of recorded event data.

The subsequent evaluation demonstrates the effectiveness, as well as the efficiency of the enhanced encoding techniques, the event reduction operations and the Hierarchical Memory Buffer data structure in a prototype implementation called OTFX.

The new enhanced encoding techniques prove to be effective and reduce memory allocation during runtime by a factor of 3.3 to 7.2 for the evaluated real-life applications and application kernels, while at the same time do not introduce any additional overhead on the tracing library. In fact, the overhead is slightly reduced since less memory must be allocated. With these results the enhanced encoding techniques outperform the memory efficiency of any existing event trace format and even well-established general purpose compression libraries, which introduce a five times higher overhead.

On the basis of theoretical models, best, worst and average case behavior are determined for the memory balancing of the Hierarchical Memory Buffer, as well as its support for the new event reduction strategies. In addition, its main parameter, the size of the internal memory bins, is evaluated and found to be best in a range of 1 - 16 KiB, which allows a good trade-off between memory efficiency and overhead. The results of the theoretical models are confirmed with empirical data from a set of real-life applications and application kernels. For the reviewed applications the Hierarchical Memory Buffer supports any event reduction operation with any grade of reduction with a maximum overhead of 5.1 % of the total library time, which equals a maximum of 0.05 % of the total measurement runtime.

Furthermore, a detailed case study of the molecular dynamics package Gromacs demonstrates that the new enhanced encoding techniques combined with the event reduction strategies and the filtering based on function duration can remarkably reduce the resulting trace size up to three orders of magnitude. This allows to keep an entire measurement within a single fix-sized memory buffer, while still providing a coarse but meaningful analysis of the application.

The enhanced encoding techniques and new event reduction strategies based on the Hierarchical Memory Buffer provide an essential improvement to event-based performance analysis by remarkably reducing the trace size during runtime while introducing minimal overhead. In addition, they allow a dynamic trace size reduction during runtime to any given memory allocation, which enables a complete in-memory event tracing workflow that meets three urgent challenges in event-based performance analysis.

Future Work

The presented enhanced encoding techniques, strategies for event reduction and the Hierarchical Memory Buffer are components of the prototype implementation OTFX, which is designed as an in-memory extension to the Open Trace Format 2. The goal is making this current prototype available to the tools that use OTF2. Because of its particular purpose, OTFX is intended as an addition to OTF2; not to replace it. Thus, to make the prototype available to tool developers, it is adapted with the full OTF2 user interface, so it can be engaged by simply linking the tool against OTFX instead of OTF2. This way, all data is recorded with the OTFX prototype during runtime but can be stored in a regular OTF2 trace archive, allowing any analysis tool that supports OTF2 to read it. In addition, to make OTFX attractive for established tools, the current research prototype must be developed into a stable, portable, and product quality implementation, as well.

A further step in this approach is a direct handover of the OTFX memory buffers to an analysis tool. For that, existing coupling techniques restricted to profiling data can be reviewed for their applicability for event tracing. These coupling techniques are an agent hierarchy based on TCP communication in Periscope [BPG10] and MALP [BPJ13] that uses the capabilities of MPI [MPI12] to connect monitoring components on multiple MPI applications to an MPI parallel analysis.

In addition, there are system specific approaches providing memory that is persistent across multiple batch jobs. The IBM Blue Gene system allows to allocate such persistent memory pages via an environment variable [IBM13]. Another trend are local non-volatile memory (NVRAM) and burst buffers [LCC+12] in the form of local solid state disks (SSD), as planned for Summit, the next leading edge HPC system at Oak Ridge National Laboratory [ORNL14]. Both, can be investigated for handing over OTFX memory buffers between measurement and analysis.
Furthermore, the Hierarchical Memory Buffer allows to apply additional strategies for event reduction or selective monitoring techniques that require hierarchical information. For this purpose the OTFX prototype supports a rapid integration of additional strategies but currently requires manual adaptions to the core implementation. For future development a designated plugin interface that allows transparent access to the hierarchical information would provide an easier way to test and deploy new strategies.

Future Research

The subsequent step in future research is the extension of the in-memory workflow enabled by the contribution of this thesis to an entire online event trace analysis workflow.
On future HPC systems targeting the exa-scale barrier monitoring and analysis need to become a fully integrated online process. On the one hand, this development is driven by the impact of decreasing memory-per-core ratio, limited I/O capabilities, and increasing core numbers of exa-scale systems on performance monitoring. On the other hand, an online event-based performance analysis workflow allows new use cases, most of which already extend tool applicability on current systems. Such use cases include the inspection of an application over a long period of time, stops on predefined conditions, altering the performance metrics to be recorded, and providing instant analysis options at any given point during the measurement.
Within an online event tracing workflow entirely omitting the file system, its tasks in a traditional event tracing workflow must be covered: coupling of measurement with analysis and the storage space for large data volumes. The first, a direct coupling of measurement and analysis other than with files in a post-mortem approach, requires a communication system to forward event information from application threads to the locations that host event trace data, that allows data exchanges for actual event analysis, and for communication with a user interface. Such a communication system must utilize scalable and hierarchical layers and must well map the data output of the measurement and input to the analysis to minimize data movement and conversion. The latter, storing large event traces in main memory, is covered by the contributions of this thesis to enable an in-memory event tracing workflow, which is an essential step towards an entire online event trace analysis workflow.

Bibliography

[ABF+10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N.R. Tallent: *HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs*. In Concurrency and Computation: Practice & Experience – Scalable Tools for High-End Computing 22(6), pages 685–701, 2010.

[AEH+11] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni: *Parallel I/O and the metadata wall*. In Proceedings of the sixth workshop on Parallel Data Storage (PDSW '11), pages 13–18, 2011.

[All14] Allinea: *Allinea MAP*. http://www.allinea.com/products/map (last visited 07 Nov 2014).

[BBC+08] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Den- neau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick: *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. 2008.

[BGWA10] D. Böhme, M. Geimer, F. Wolf, L. Arnold: *Identifying the root causes of wait states in large-scale parallel applications*. In Proceedings of the 39th International Conference on Parallel Processing (ICPP), pages 90–100, 2010.

[BHJH10] Holger Brunst, Daniel Hackenberg, Guido Juckeland, and Heide Rohling: *Comprehensive Performance Tracking with Vampir 7*. In Tools for High Performance Computing 2009, pages 17–29, 2010.

[BLBS92] David H. Bailey, Leonardo Dagum, Eric Barszcz, and Horst D. Simon: *NAS Parallel Benchmark Results*. In IEEE Parallel and Distributed Technology, 1992.

[BM07] A. R. Bernat and B. P. Miller: *Incremental Call-path Profiling*. In Concurrency and Computation: Practice and Experience 19(11), pages 1533–1547, 2007.

[BM11] Andrew R. Bernat and Barton P. Miller: *Anywhere, Any Time Binary Instrumentation*. ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), 2011.

[BMSB03] Holger Brunst, Allen D. Malony, Sameer S. Shende, and Robert Bell: *Online Remote Trace Analysis of Parallel Applications on High Performance Clusters*. In High Performance Computing, LNCS 2858, pages 440–449, 2003.

[BNM03] Holger Brunst, Wolgang E. Nagel, and Allen D. Malony *A Distributed Performance Analysis Architecture for Clusters*. In Proceedings of IEEE International Conference on Cluster Computing, pages 73–81, 2003.

[BN03] H. Brunst and W. E. Nagel: *Scalable Performance Analysis of Parallel Systems: Concepts and Experiences*. In Parallel Computing: Software, Algorithms, Architectures Applications, pages 737–744, 2003.

[Boe14] David Böhme: *Characterizing Load and Communication Imbalance in Parallel Applications*. (Ph.D. thesis) In IAS Series 23, ISBN 978-3-89336-940-9, 2014.

[BPG10] S. Benedict, V. Petkov, and M. Gerndt: *PERISCOPE: An Online-Based Distributed Performance Analysis Tool*. In Tools for High Performance Computing 2009, pages 1–16, 2010.

[BPJ13] J.-B. Besnard, M. Pérache, and W. Jalby: *Event Streaming for Online Performance Measurements Reduction*. In Proceedings of the 42nd International Conference on Parallel Processing, 2013.

[Bru08] Holger Brunst: *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs*., 2008.

[BS04] H.M. Blackburn, S.J. Sherwin: *Formulation of a Galerkin Spectral Element Fourier Method for Three-dimensional Incompressible Flows in Cylindrical Geometries*. In Journal of Computational Physics 197(2), pages 759–778, 2004.

[BSG+12] D. Böhme, B.R. de Supinski, M.Geimer, M.Schulz, F.Wolf: *Scalable Critical-Path Based Performance Analysis*. In Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), pages 1330–1340, 2012.

[BSC10] Barcelona Supercomputing Center: *Paraver Internals and Details*.
http://www.bsc.es/computer-sciences/performance-tools/documentation
(last visited 13 Nov 2014)

[BSC14] Barcelona Supercomputing Center: *Extrae User Guide Manual for Version 2.5.1*.
http://www.bsc.es/computer-sciences/performance-tools/documentation
(last visited 13 Nov 2014)

[BWNH01] H. Brunst, M. Winkler, W. E. Nagel and H.-C. Hoppe: *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach*. In International Conference on Computational Science (ICCS), pages 751–760, 2001.

[C11] *Standard C Library Documentation*. In ISO/IEC 9899:2011, 2011.

[CBL07] M. Casas, R. M. Badia, and J. Labarta: *Automatic phase detection of MPI application*. In Proceedings of the Conference on Parallel Computing (ParCo), Advances in Parallel Computing 15, IOS Press, pages 129–136, 2007.

[CEP01a] CEPBA (European Center for Parallelism of Barcelona): *Paraver Version 3.0 Tracefile Description*.
http://www.bsc.es/computer-sciences/performance-tools/documentation
(last visited 13 Nov 2014)

[CEP01b] CEPBA (European Center for Parallelism of Barcelona): *Paraver Version 3.1 Reference Manual*.
http://www.bsc.es/computer-sciences/performance-tools/documentation
(last visited 13 Nov 2014)

[CEP14] CEPBA (European Center for Parallelism of Barcelona): *Introduction to Dimemas*.
http://www.bsc.es/computer-sciences/performance-tools/documentation
(last visited 13 Nov 2014)

[CRE14] *CRESTA: Collaborative Research into Exascale Systemware, Tools and Applications*.
http://cresta-project.eu (last visited 29 July 2014)

[DG96] Peter Deutsch and Jean-Loup Gailly: *ZLIB Compressed Data Format Specification Version 3.3*. 1996.

[DKMN08] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel: *Internal Timer Synchronization for Parallel Event Tracing*. In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 202–209, 2008.

[EWG+12] 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 Applications, Tools and Techniques on the Road to Exascale Computing, Advances in Parallel Computing 22, pages 481–490, 2012.

[FWP09] W. Frings, F. Wolf, and V. Petkov: *Scalable Massively Parallel I/O to Task-Local Files*. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09, ACM, pages 17:1—17:11, 2009.

[FWS10] K. Fuerlinger, N.J. Wright, and D. Skinner: *Effective Performance Measurement at Petascale Using IPM*. In Proceedings of The Sixteenth IEEE International Conference on Parallel and Distributed Systems, 2010.

[Gei13] Markus Geimer: *The Scalasca Performance Analysis Toolset*. Presentation at SEA Conference'13, Boulder, USA, http://sea.ucar.edu/sites/default/files/Geimer_Scalasca.pdf (last visited 25 Nov 2014)

[Ghe08] Sanjay Ghemawat: *Documentation for the CPU profiler*. Within the gperftools 2.2.90 package (last modified 2008).
http://google-perftools.googlecode.com (last download 07 Nov 2014)

[GG14] Google *Google Performance Tools*.
http://google-perftools.googlecode.com (last visited 07 Nov 2014)

[GKM82] S.L. Graham, P.B. Kessler, and M.K. Mckusick: *Gprof: A Call Graph Execution Profiler*. In Proceedings of the 1982 SIGPLAN symposium on compiler construction, pages 120–126, ACM 1982.

[GRO14] *GROMACS Website*.
http://www.gromacs.org (last visited 29 July 2014)

[GSS+12] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie: *Further Improving the Scalability of the Scalasca Toolset*. In Proceedings of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing, pages 463–474, 2012.

[GWT14] GWT-TUD GmbH: *Vampir 8 User Manual*.
http://www.vampir.eu/tutorial/manual (last visited 12 Nov 2014)

[GWW+10] Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr: *The Scalasca Performance Toolset Architecture*. In Concurrency and Computation: Practice and Experience 22(6) pages 702–719, 2010.

[HKS08] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl: *GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation*. In Journal of Chemical Theory and Computation 4(3), pages 435–447, 2008.

[HMC94] J. K. Hollingsworth, B. P. Miller and J. Cargille: *Dynamic Program Instrumentation for Scalable Performance Tools*. In Proceedings of Scalable High Performance Computing Conference, 1994.

[HSL10] T. Hoefler, T. Schneider, and A. Lumsdaine: *Characterizing the Influence of System Noise on Large-Scale Applications by Simulation*. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2010.

[IBM13] IBM, International Technical Support Organization: *IBM System Blue Gene Solution: Blue Gene/Q Application Development*. In IBM Redbooks,
http://www.ibm.com/redbooks (last visited 11 Dec 2014)

[Int14a] Intel Corporation: *Intel VTune Amplifier 2015*.
http://software.intel.com/en-us/intel-vtune-amplifier-xe (last visited 10 Nov 2014).

[Int14b] Intel Corporation: *Intel® Trace Analyzer Reference Manual*.
http://software.intel.com/en-us/intel-software-technical-documentation
(last visited 11 Nov 2014)

[ISC+12] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole: *Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware*. In Proceedings of the 21th International Symposium on High Performance Distributed Computing, HPDC'12, ACM, pages 49—60, 2012.

[Jai91] Raj Jain: *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, 1991.

[KBB+06] A. Knüpfer, R. Brendel, H. Brunst, H. Mix and W. E. Nagel: *Introducing the Open Trace Format (OTF)*. In 6th International Conference for Computational Science (ICCS), pages 526–533, 2006.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel: *The Vampir Performance Analysis Tool Set*. In Tools for High Performance Computing, pages 139–155, Springer, 2008.

[KN05a] Andreas Knüpfer and Wolfgang E. Nagel: *Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis*. In Proceedings of the 2005 International Conference on Parallel Processing, pages 165–172, 2005.

[KN05b] Andreas Knüpfer and Wolfgang Nagel: *New Algorithms for Performance Trace Analysis Based on Compressed Complete Call Graphs*. In Proceedings of the International Conference on Computational Science (ICCS), pages 7–36, 2005.

[KN06] Andreas Knüpfer and Wolfgang E. Nagel: *Compressible Memory Data Structures for Event-Based Trace Analysis*. In Future Generation Computer Systems 22(3), pages 359–368, 2006.

[Knu08] Andreas Knüpfer: *Advanced Memory Data Structures for Scalable Event Trace Analysis*. (Ph.D. thesis), 2008.

[KRM+12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf: *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.

[LCC+12] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn: *On the role of burst buffers in leadership-class storage systems*. In Proceedings of the 2012 IEEE Conference on Massive Data Storage, 2012.

[LCM+00] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen: *A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates*. In Proceedings of SC2000: High Performance Networking and Computing Conference, 2000.

[LCS+11] G. Llort, M. Casas, H. Servat; K. Huck, J. Gimenez, J. Labarta: *Trace Spectral Analysis toward Dynamic Levels of Detail*. In 17th International Conference on Parallel and Distributed Systems (ICPADS), pages 332–339, 2011.

[LDTW14] Daniel Lorenz, Robert Dietrich, Ronny Tschüter, and Felix Wolf: *A Comparison between OPARI2 and the OpenMP Tools Interface in the Context of Score-P*. In Proceedings of the 10th International Workshop on OpenMP (IWOMP), pages 161–172, 2014.

[LGS+10] G. Llort, J. González, H. Servat, J. Giménez, and J. Labarta: *On-line Detection of Large-scale Parallel Application's Structure*. In 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.

[LGW+12] Matthias Lieber, Verena Grützun, Ralf Wolke, Matthias S. Müller, and Wolfgang E. Nagel: *Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4*. In Applied Parallel and Scientific Computing, LNCS 7133, pages 131–141, Springer, 2012.

[MCA+14] John Mellor-Crummey, Laksono Adhianto, Mike Fagan, Mark Krentel, and Nathan Tallent: *HPCToolkit User's Manual*.
http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf (last visited 10 Nov 2014)

[MCC+95] 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*. In IEEE Computer 28(11), Special issue on performance evaluation tools for parallel and distributed computer systems, pages 37–46, 1995.

[MCLD01] S. Moore, D. Cronk, K. London, and J. Dongarra: *Review of Performance Analysis Tools for MPI Parallel Programs*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, LNCS 2131, pages 241–248, 2001.

[MDG+04] J. Michalakes, J. Dudhia, D. Gill, T.B. Henderson, J. Klemp, W. Skamarock, and W. Wang: *The Weather Research and Forecast Model: Software Architecture and Performance*. In 11th ECMWF Workshop on the Use of High Performance Computing in Meteorology, 2004.

[MK09] K. Mohror and K. L. Karavanic: *Evaluating Similarity-based Trace Reduction Techniques for Scalable Performance Analysis*. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pages 55:1—55:12, 2009.

[MKJ+07] 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 Advances in Parallel Computing 15: Parallel Computing: Architectures, Algorithms and Applications, pages 637–644, 2007.

[MLW11] J. Mußler, D. Lorenz, and F. Wolf: *Reducing the Overhead of Direct Application Instrumentation Using Prior Static Analysis*. In Proceedings of the 17th Euro-Par Conference, LNCS 6852, pages 65–76, 2011.

[MML+11] Henry Markram, Karlheinz Meier,Thomas Lippert, Sten Grillner, Richard Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria: *Introducing the Human Brain Project*. In Procedia Computer Science 7, pages 39–42, 2011.

[Moo65] Gordon E. Moore: *Cramming More Components onto Integrated Circuits*. In Electronics 38(8), pages 114–117, 1965.

[MPI12] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 3.0*. 2012.
http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (last visited 18 Nov 2014)

[MSM05] A. D. Malony, S. S. Shende, and A. Morris: *Phase-Based Parallel Performance Profiling*. In Proceedings of the ParCo'05 conference, Parallel Computing: Current and Future Issues of High-End Computing, 2005.

[MWL+07] Matthias S. Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, Carl Ponder: *SPEC MPI2007 – An Application Benchmark Suite for Parallel Systems Using MPI*. In Concurrency and Computation: Practice and Experience 22(2), pages 191–205, 2010.

[NAW+96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach: *VAMPIR: Visualization and Analysis of MPI Resources*. In Supercomputer 1, pages 69–80, 1996.

[NMM+08] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller: *A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet*. International Workshop on Scalable Tools for High-End Computing (STHEC 2008), 2008.

[NRM+09] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B.R. de Supinski: *ScalaTrace: Scalable Compression and Replay of Communication Traces for High-performance Computing*. In Journal of Parallel and Distributed Computing 69(8), pages 696–710, 2009

[NRR97] O. Y. Nickolayev, P. C. Roth, D. A. Reed: *Real-Time Statistical Clustering for Event Trace Reduction*. In the International Journal of Supercomputer Applications and High Performance Computing 11(2), pages 144–159, 1997.

[ORNL14] Scott Jones: *Oak Ridge to Acquire Next Generation Supercomputer*.
https://www.olcf.ornl.gov/2014/11/14/oak-ridge-to-acquire-next-generation-supercomputer (last visited 11 Dec 2014)

[OTF14] *Open Trace Format 2 User Manual*. 2014.

[Rei05] J. Reinders: *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[SBS08] M. Schulz, G. Bronevetsky, and B. R. Supinski: *On the Performance of Transparent MPI Piggyback Messages*. In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 194–201, 2008.

[SDT14a] Scalasca Development Team *Scalasca 2.1 | User Guide*. http://www.scalasca.org/software/scalasca-2.x/ (last visited 13 Nov 2014)

[SDT14b] Scalasca Development Team *CUBE 4.2.3 – User Guide*. http://www.scalasca.org/software/cube-4.x/ (last visited 13 Nov 2014)

[SDT14c] Scalasca Development Team *Scalasca 2.1 Performance Properties*. http://www.scalasca.org/software/scalasca-2.x/ (last visited 13 Nov 2014)

[SGM+08] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford: *Open | SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis*. In Scientific Programming 16(2), pages 105–121, 2008.

[SH02] F. Schmuck and R. Haskin: *GPFS: A Shared-Disk File System for Large Computing Clusters*. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages 231—244, 2002.

[SILC09] *SILC Project*. http://www.vi-hps.org/projects/silc/ (last visited 12 Nov 2014)

[SM06] Sameer Shende and Allen D. Malony: *The TAU Parallel Performance System*. In International Journal of High Performance Computing Applications 20(2), pages 287–331, 2006.

[SLGL10] H. Servat, G. Llort, J. Giménez, J. Labarta: *Detailed Performance Analysis Using Coarse Grain Sampling*. In Euro-Par 2009 - Parallel Processing Workshops, pages 185–198, 2010.

[SL11] Senthil Shanmugasundaram and Robert Lourdusamy: *A Comparative Study of Text Compression Algorithms*. International Journal of Wisdom Based Computing 1(3), pages 68–76, 2011.

[Sun08] Sun Microsystems, Inc.: *Lustre File System – High-Performance Storage Architecture and Scalable Cluster File System*. 2008.

[SWW09] Zoltán Szebenyi, Felix Wolf, and Brian J.N. Wylie: *Space-Efficient Time-Series Call-Path Profiling of Parallel Applications*. Proceedings of the ACM/IEEE Conference on Supercomputing (SC09), ACM, 2009.

[TAU12a] *TAU User Guides*. 2012. http://www.cs.uoregon.edu/research/tau/docs.php (last visited 14 Nov 2014)

[TAU12b] *TAU Reference Guide*. 2012. http://www.cs.uoregon.edu/research/tau/docs.php (last visited 14 Nov 2014)

[TMCF09] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan: *Binary Analysis for Measurement and Attribution of Program Performance*. In PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 441–452, 2009.

[Top14] Top500: *Top 500 Supercomputer Sites – November 2014 list*. http://www.top500.org/ (last visited 25 Nov 2014)

[VM01] J.S. Vetter and M.O. McCracken: *Statistical Scalability Analysis of Communication Operations in Distributed Applications*. In Proceeding of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), 2001.

[WBB12] Matthias Weber, Ronny Brendel, and Holger Brunst: *Trace File Comparison with a Hierarchical Sequence Alignment Algorithm*. In Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, pages 247–254, 2012.

[WDKN13] Michael Wagner, Jens Doleschal, Andreas Knüpfer, and Wolfgang E. Nagel: *Runtime Message Uniquification for Accurate Communication Analysis on Incomplete MPI Event Traces*. In Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13), pages 123–128, ACM 2013.

[WDKN14] Michael Wagner, Jens Doleschal, Andreas Knüpfer, and Wolfgang E. Nagel: *Selective Runtime Monitoring: Non-intrusive Elimination of High-frequency Functions*. In Proceedings of the International Conference on High Performance Computing & Simulation (HPCS), pages 295–302, 2014.

[Wea14] M. Weaver: *Linux perf_event Features and Overhead*. In Proceedings of the 2013 FastPath Workshop, 2013.

[WGM+10] Brian J. N. Wylie, Markus Geimer, Bernd Mohr, David Böhme, Zoltán Szebenyi, and Felix Wolf: *Large-scale Performance Analysis of Sweep3D with the Scalasca Toolset*. In Parallel Processing Letters, 20(4), pages 397—414, 2010.

[WHB14] Michael Wagner, Tobias Hilbrich, and Holger Brunst: *Online Performance Analysis: An Event-based Workflow Design Towards Exascale*. In Proceedings of HPCC 2014, 2014.

[WKN12] Michael Wagner, Andreas Knüpfer, and Wolfgang E. Nagel: *Enhanced Encoding Techniques for the Open Trace Format 2*. In Procedia Computer Science 9, pages 1979–1987, 2012.

[WKN13] Michael Wagner, Andreas Knüpfer, and Wolfgang E. Nagel: *Hierarchical Memory Buffering Techniques for an In-Memory Event Tracing Extension to the Open Trace Format 2*. In Proceedings of the 42nd International Conference on Parallel Processing, pages 970–976, 2013.

[WM03] Felix Wolf and Bernd Mohr: *KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Applications*. In Proceedings of European Conference on Parallel Computing (Euro-Par), LNCS 2790, pages 1301–1304, 2003.

[WM04] Felix Wolf and Bernd Mohr: *EPILOG Binary Trace-Data Format*. Technical Report FZJ-ZAM-IB-2004-06, 2004.

[WMS+13] Matthias Weber, Kathryn Mohror, Martin Schulz, Bronis R. de Supinski, Holger Brunst, and Wolfgang E. Nagel: *Alignment-Based Metrics for Trace Comparison*. In Euro-Par 2013 Parallel Processing, pages 29–40, 2013.

[WN13] Michael Wagner and Wolfgang E. Nagel: *Strategies for Real-Time Event Reduction*. In Euro-Par 2012: Parallel Processing Workshops, LNCS 7640, pages 429–438, Springer 2013.

[ZIH14] Center for Information Services and High Performance Computing (ZIH): *VampirTrace 5.14.4 User Manual*. http://www.tu-dresden.de/zih/vampirtrace (last visited 12 Nov 2014)

[ZL77] Jacob Ziv and Abraham Lempel: *A Universal Algorithm for Sequential Data Compression*. In IEEE Transactions on Information Theory 23(3), pages 337–343, 1977.

[ZLGS99] Omer Zaki, Ewing Lusk, William Gropp, Deborah Swider: *Toward Scalable Performance Visualization with Jumpshot*. In International Journal of High Performance Computing Applications 13(3), pages 277–288, 1999.

List of Figures

2.1 Optimization cycle 5
2.2 The three stages of performance analysis 6
2.3 Static and dynamic load imbalance with timelines and profiles 8
2.4 Classification of performance analysis tools. 9
2.5 Vampir/VampirSever architecture 11
2.6 Vampir global timeline 13
2.7 Vampir process timeline 13
2.8 Vampir function summary 14
2.9 Vampir communication matrix 14
2.10 Vampir custom display arrangement 15
2.11 Vampir performance radar 15
2.12 Paraver timeline view 16
2.13 Paraver textual view 17
2.14 Paraver statistics view 17
2.15 Late sender/receiver pattern 18
2.16 Cube display 19
2.17 Cube display with system topology 19
2.18 Paraprof function summary 20
2.19 Paraprof 3D visualization 20
2.20 Score-P tool architecture 21
2.21 OTF2 archive layout 23
2.22 Successive Compression in a CCG 26
2.23 Bias on parallel behavior due to intermediate memory buffer flushes 29

3.1 Three steps for in-memory event tracing 34
3.2 Basic memory representation of event records 37
3.3 Memory representation of an exemplary event sequence 37
3.4 Splitting of timing information and event data for a generic event record 38
3.5 Splitting of timing information and event data for the exemplary sequence 38
3.6 Leading zero elimination for a generic event record 39
3.7 Delta encoding for the exemplary event sequence 39
3.8 Merging of token and number of remaining data bytes of the first attribute 40
3.9 Distribution of event classes by number of occurrences 41
3.10 Merging of token and number of remaining data bytes of the first attribute 41
3.11 Merging of token and region ID for enter events 42
3.12 Number of different region IDs 42

3.13 Average event rates of the reviewed applications and kernels. 43
3.14 Distribution of event classes by size of memory allocation 46
3.15 Correlation between cause and impact in a basic timeline visualization 47
3.16 Callstack distribution for selected applications 48
3.17 Callstack distribution for all SPEC MPI 2007 and NAS Parallel Benchmarks applications 49
3.18 Duration distribution for selected applications and kernels 51

4.1 Event reduction with a flat continuous memory buffer 56
4.2 Flat partitioned event representation 56
4.3 Event reduction with a hierarchical event representation 58
4.4 Illustration of a two-dimensional Hierarchical Memory Buffer data structure 61
4.5 Algorithms to alter the Hierarchical Memory Buffer 62
4.6 Algorithm to assign memory bins 62
4.7 Construction of the Hierarchical Memory Buffer (Part I) 63
4.7 Construction of the Hierarchical Memory Buffer (Part II) 64
4.8 Cyclic segmented buffer layout with Hierarchical Memory Sub-buffers 65
4.9 Algorithm to revoke all memory bins for a given event class and call stack level 66
4.10 Algorithm to reduce an complete event class 66
4.11 Reduction by event class on the Hierarchical Memory Buffer 67
4.12 Algorithm to reduce deepest call stack level 68
4.13 Reduction by calling depth on the Hierarchical Memory Buffer (Part I) 68
4.13 Reduction by calling depth on the Hierarchical Memory Buffer (Part II) 69
4.14 Algorithms to alter the Hierarchical Memory Buffer and skip short code regions 70
4.15 Algorithm to initialize the event queue 72
4.16 Algorithm to read next event 72
4.17 Algorithm to initialize event queue with additional hierarchy criterion 74
4.18 A communication pattern of three successive MPI send/receive calls 77
4.19 A communication pattern with missing MPI events 78
4.20 A communication pattern with missing MPI events and new message matching 80
4.21 Distribution of samples based on powers of two 82

5.1 Memory allocation for different encoding techniques 88
5.2 Average time error for different timer resolutions 90
5.3 Memory allocation for different trace data formats 91
5.4 Memory allocation for different trace data formats in detail 92
5.5 Runtime overhead for different trace data formats 93
5.6 Number of events for different memory bin sizes (modeled) 95
5.7 Number of events for different memory bin sizes (32 MiB buffer) 96
5.8 Number of events for different memory bin sizes (different buffer sizes) 96
5.9 Runtime overhead for different memory bin sizes 97
5.10 Runtime overhead for different memory bin sizes and buffer sizes 99
5.11 Runtime overhead for code region reduction 100
5.12 Event trace visualization of code region reduction 102

5.13 Time to read events depending on the maximum calling depth 103
5.14 Vampir screenshot displaying the distribution of buffer flushes 108
5.15 Vampir screenshot displaying the distribution of buffer flushes and MPI wait times . . . 108
5.16 Overhead caused by intermediate buffer flushes 109
5.17 Distribution of code regions by time and calling depth 110
5.18 Event trace visualization of reduction to calling depth five 112
5.19 Event trace visualization of reduction to calling depth five (zoomed) 112
5.20 Automatic analysis results of Scalasca in Cube without reduction 113
5.21 Automatic analysis results of Scalasca in Cube with reduction 113

List of Tables

4.1 Complexity of basic operators for the time iterator 73

5.1 Overview of evaluated applications and benchmarks 86
5.2 Reduction of memory allocation for each encoding technique 88
5.3 Runtime overhead and memory allocation of OTF, OTF2, OTF2 with zlib, and OTFX . . 93
5.4 Average event trace sizes per process with and without code region reduction 101
5.5 Number of messages per message envelope and per location 105
5.6 Number of different message envelopes per location 105
5.7 Additional memory requirements per location 106
5.8 Trace sizes for different reduction strategies 110
5.9 Trace sizes for different reduction strategies with function inlining 111