Advanced Memory Data Structures for Scalable Event Trace Analysis

Dissertation

zur Erlangung des akademischen Grades Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Mathematiker Andreas Knüpfer
geboren am 25. Juli 1976 in Zwickau

Gutachter:
Prof. Dr. rer. nat. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr.-Ing. Wolfgang Lehner, Technische Universität Dresden
Prof. Dr. techn. habil. Dieter Kranzlmüller, Ludwig-Maximilians-Universität München

Tag der Verteidigung: 16. Dezember 2008

Dresden im März 2009

Contents

Abstract 5

1 Performance Analysis for HPC Applications 7

1.1 Computation and Performance . . . 7
1.2 Performance Analysis and Optimization . . . 8
1.3 Performance Analysis Tools . . . 8
1.4 Contribution of this Thesis . . . 9

2 State-of-the-Art in Event-Based Trace Analysis and Related Work 11

2.1 Tracing and Logging . . . 11
2.2 Trace Analysis Tools . . . 12
2.2.1 Vampir and VampirServer . . . 12
2.2.2 Paraver and Dimemas . . . 17
2.2.3 Kojak and Scalasca . . . 18
2.2.4 Jumpshot . . . 22
2.2.5 Debugging Wizard (DeWiz) . . . 22
2.2.6 Tuning and Analysis Utilities (TAU) . . . 26
2.3 Trace File Formats . . . 28
2.3.1 Common Design of Trace File Formats . . . 28
2.3.2 The Vampir Trace Format Version 3 (VTF3) . . . 30
2.3.3 The Structured Trace Format (STF) . . . 30
2.3.4 The Open Trace Format (OTF) . . . 31
2.3.5 The Epilog Trace Format . . . 32
2.3.6 The Jumpshot Trace Formats . . . 33
2.3.7 The Paraver Trace Format . . . 33
2.3.8 The DeWiz Trace Format . . . 34
2.3.9 The TAU Trace Format . . . 34
2.4 Memory Data Structures . . . 35
2.4.1 The Vampir and VampirServer Data Structures . . . 35
2.4.2 The EARL Data Structures . . . 36
2.4.3 The DeWiz Data Structures . . . 37
2.4.4 The TAU Data Structures . . . 38
2.4.5 Similarity to Trace File Formats . . . 39
2.5 Access Methods to Event Data Structures . . . 39
2.5.1 Sequential Iterator . . . 39
2.5.2 Time Interval Search . . . 40
2.5.3 Statistic Summaries . . . 40
2.5.4 Timeline Visualization . . . 42
2.5.5 Automatic Analysis . . . 44
2.6 Event Trace Compression by Statistical Clustering . . . 45
2.7 Memory Access Traces and Compression . . . 46
2.8 Compression of MPI Replay Traces . . . 47

3 The Design of the CCG Data Structure 49
3.1 Trace Data and Trace Information 49
3.2 Tree Data Structures for Event Traces 50
  3.2.1 Time Stamps versus Time Durations 52
  3.2.2 The Bounded Branching Factor 52
3.3 In-Memory Compression 53
  3.3.1 Deviation Bounds for Soft Properties 54
  3.3.2 Sub-Tree Comparison 55
  3.3.3 Computational Effort 56
3.4 Customized Analysis Algorithms 57
  3.4.1 The Conservative Approach 58
  3.4.2 Event Access, Iteration and Searching 58
  3.4.3 The Cached Summary Query 59
  3.4.4 The Timeline Query 60
  3.4.5 MPI Send-Receive Matching 61
  3.4.6 MPI Collective Operation Matching 62
  3.4.7 Automatic Analysis Methods 63
3.5 Persistent Storage and Restoring 63
3.6 Summary 64

4 Algorithms for the CCG Data Structure 65
4.1 CCG Construction 65
  4.1.1 General Construction 65
  4.1.2 Splitting of Wide Nodes 66
  4.1.3 Graph Node Encoding 69
  4.1.4 Graph Node Allocation 71
4.2 CCG Compression 71
  4.2.1 Search for Replacement Nodes 72
  4.2.2 Caching of Nodes 72
  4.2.3 Influence of the Node Search Order 74
  4.2.4 Node Comparison 75
  4.2.5 Compression Metrics 77
4.3 The Combined Construction and Compression Algorithm 78
4.4 Advanced Construction and Compression Techniques 79
  4.4.1 Re-Compression of Existing CCGs 79
  4.4.2 Merging of Disjoint CCGs 82
  4.4.3 Adaptive Deviation Bounds 82
4.5 CCG Analysis Algorithms 85
  4.5.1 The Random-Access Iterator 85
  4.5.2 Timestamp Search 86
  4.5.3 Summary Query Algorithms 87
  4.5.4 The Timeline-Rendering Algorithm 90
  4.5.5 The MPI Send-Receive Matching Algorithm 91
4.6 Persistent Storage and Restoring 92
4.7 CCGs with Distributed Data 94
  4.7.1 Distributed Data Decomposition 95
  4.7.2 Distributed CCG Construction 95
  4.7.3 Distributed CCG Compression 96
  4.7.4 Distributed Serialization and Restoring 97
  4.7.5 Distributed Evaluation 97

5 Evaluation of CCG Algorithms **99**
5.1 Theoretical and Synthetic Evaluation . . . 99
5.1.1 Compression Model . . . 99
5.1.2 Non-Monotone Compression . . . 100
5.1.3 Best Case Compression . . . 101
5.1.4 Worst Case Compression . . . 102
5.2 Real-World Construction and Compression . . . 104
5.2.1 Small Scale Compression . . . 105
5.2.2 Large Scale Compression . . . 108
5.2.3 Influence of Branching Factor . . . 108
5.2.4 Influence of Trace Size . . . 109
5.2.5 Influence of Search Length Parameter . . . 114
5.3 Advanced Construction and Compression Algorithms . . . 116
5.3.1 Re-Compression of CCGs . . . 116
5.3.2 Adaptive CCG Compression . . . 117
5.3.3 Serialization and Restore of CCGs . . . 118
5.4 Cached Summary Queries . . . 119
5.4.1 Cache Strategies . . . 119
5.4.2 Experiment Results . . . 120
5.5 MPI Send-Receive Matching . . . 120
5.6 Recommended Parameter Settings . . . 123
5.6.1 Deviation Bounds for Soft Properties . . . 123
5.6.2 Algorithm Parameters . . . 125
5.7 Comparison to State-of-the-Art Tools . . . 125

6 Conclusion and Outlook **129**

A Acknowledgements **131**

List of Figures **133**

Bibliography **137**

Abstract

The thesis presents a contribution to the analysis and visualization of computational performance based on event traces with a particular focus on parallel programs and High Performance Computing (HPC).

Event traces contain detailed information about specified incidents (events) during run-time of programs and allow minute investigation of dynamic program behavior, various performance metrics, and possible causes of performance flaws. Due to long running and highly parallel programs and very fine detail resolutions, event traces can accumulate huge amounts of data which become a challenge for interactive as well as automatic analysis and visualization tools.

The thesis proposes a method of exploiting redundancy in the event traces in order to reduce the memory requirements and the computational complexity of event trace analysis. The sources of redundancy are repeated segments of the original program, either through iterative or recursive algorithms or through SPMD-style parallel programs, which produce equal or similar repeated event sequences.

The data reduction technique is based on the novel *Complete Call Graph* (CCG) data structure which allows domain specific data compression for event traces in a combination of lossless and lossy methods. All deviations due to lossy data compression can be controlled by constant bounds. The compression of the CCG data structure is incorporated in the construction process, such that at no point substantial uncompressed parts have to be stored. Experiments with real-world example traces reveal the potential for very high data compression. The results range from factors of 3 to 15 for small scale compression with minimum deviation of the data to factors > 100 for large scale compression with moderate deviation.

Based on the CCG data structure, new algorithms for the most common evaluation and analysis methods for event traces are presented, which require no explicit decompression. By avoiding repeated evaluation of formerly redundant event sequences, the computational effort of the new algorithms can be reduced in the same extent as memory consumption.

The thesis includes a comprehensive discussion of the state-of-the-art and related work, a detailed presentation of the design of the CCG data structure, an elaborate description of algorithms for construction, compression, and analysis of CCGs, and an extensive experimental validation of all components.

1 Performance Analysis for HPC Applications

This chapter provides an introduction to the field of High Performance Computing (HPC) in general and the requirement for performance optimization in this domain. Furthermore, it explains the contribution of performance measurement and analysis in the optimization process and shows the contribution of this thesis for the efficient analysis of huge amounts of trace data.

1.1 Computation and Performance

Computation has always been an important tool in science and engineering. It allows to deduce accurate and specific facts about a real-world system from general scientific theories by means of mathematical models. This intellectual tool itself is much older than the term *computer* as we use it today. Indeed, *computer* used to be a human profession in former times and the modern English word for a *computing machine* was derived from that [Gri05]. Interestingly enough, parallel and distributed computing was already prevalent in this past era!

After the birth of electronic computing in the middle of the 20'th century, this scientific tool has become much more powerful and more affordable and was utilized in more and more scientific and engineering domains. The computational speed has been growing enormously in an exponential manner for almost 50 years according to Moore's Law [Moo65], and the growth of the memory and storage capacities has been almost as fast. This has created unpredicted advancements in all fields of science and engineering and far beyond. Today, computing devices are regarded as ubiquitous and omnipresent everyday tools.

Today, computational simulation is perfectly accepted as the third cornerstone of scientific methodology besides theory and experiment. But despite the tremendous growth in the available computing power, performance is still regarded critical. This is due to the fact that the steady improvement is leveraging more and better utilization. On one hand, existing simulations can be improved through better computing resources by using more detailed resolution or more complex models. This has driven the evolution of weather and climate simulations since their origins, for example. On the other hand, whole new application domains became feasible that were considered impossible before, for example computational genetics and biology in the last decade.

Furthermore, the scientific communities anticipate the improving computing power in order to generate scientific advancements. The same is true for engineering achievements through computation. Therefore, efficiency has always played an important role in scientific computation and always will.

The field of High Performance Computing (HPC) is dedicated to the enhancement of computing performance. On one hand, this includes hardware design, ranging from developing special purpose processors to composing commodity components. On the other hand, it contains algorithmic improvements and software optimization in order to utilize the available computing resources to a maximum possible extend for the benefit of the target application.

1.2 Performance Analysis and Optimization

The subject of performance optimization is the enhancement of computational efficiency of existing application software. It also includes algorithm optimization, but usually this aspect is determined during the earlier software design phase.

Primarily, performance optimization attempts to increase efficiency by adapting the software to the particular hardware platform. This is by no means a trivial task. First of all, because there will be no perfect consistency between the hardware's provisions and the software's requirements. Furthermore, todays hardware platforms are very complex as a result of past performance improvements. This includes complex memory access via multi-level caches, pipelined instruction execution with complicated dependencies, branch prediction, speculative execution, and last but not least parallel execution.

All of those concepts enable a considerable performance gain when used properly. Yet, counteracting any one of those concepts will reliably prevent achieving a notable share of the peak performance. In addition, it will inhibit the benefits of further hardware improvements. Optimizations may include modifications that are profitable for all or many platforms, for example increasing the locality of memory accesses, or only for a selected platform, for example adjustment to a particular size of the second level cache.

The contribution of performance measurement and analysis to the optimization process is threefold: On one hand, it determines the components of a complex software system that are essential for the over-all performance, because they consume a large part of the execution time. On the other hand, it identifies the parts that are worthwhile for optimization because their actual performance is much lower than anticipated (regarding any performance property). And finally, the analysis can uncover causes of insufficient performance and opportunities for optimization.

All three phases require detailed insight in the complex execution behavior of the target software on a particular hardware platform. This includes sequential as well as parallel behavior from arithmetic operations, memory accesses, and input/output to communication, synchronization, and load balancing.

1.3 Performance Analysis Tools

Software tools for performance measurement and analysis provide the user with the detailed performance information that is the basis for successful optimization of a program. Almost all tools separate the measurement part from the analysis part, in order to allow offline analysis of the performance behavior and to minimize the overhead during run time.

There are two major paradigms for performance measurement. The first and most widely known one is *profiling* which collects aggregated summary information for selected components of a program. This is achieved either in a deterministic or a stochastic manner. The latter is also known as *sampling*.

The second paradigm is *event tracing* which records detailed protocols of individual events and their properties, where the *events* are selected points of interest during the execution.

The profiling approach causes only a moderate measurement overhead and produces a result data set of limited size, which limits the detail resolution at the same time. Tracing, is able to produce very detailed data with almost arbitrary resolution and may therefore produce enormous amounts of data. This imposes a challenge for interactive visualization and analysis of event traces.

Existing solutions either avoid creating oversized traces or use distributed storage and parallel analysis in order to cope with the enormous amounts of trace data. This thesis presents an alternative method for the storage and the analysis of huge amounts of event trace data.

1.4 Contribution of this Thesis

The proposed approach aims to compress the event trace data without loss of relevant information. This comprises of two components: Firstly, the detection and removal of redundancy due to repetitions in the event streams. And secondly, filtering of irrelevant fluctuations in certain properties of the events.

The former is not unlike lossless data compression. It exploits repeated subsequences of events that are very frequently found in large parallel traces. Repetitions originate either from SPMD-style (Single Program Multiple Data) parallel programs (spatial repetitions) or from iterative algorithms and even recursive algorithms (temporal repetitions). The latter component is related to lossy data compression. It filters irrelevant fluctuations below an adjustable threshold. This enhances the regularity of the event data which is beneficial for the first stage of compression.

The newly presented method provides a new memory data structure for event trace data called *Complete Call Graphs* (CCG) which allows domain specific data compression without the requirement for explicit decompression. Furthermore, it provides appropriate algorithms for all common analysis tasks.

The chapters of this thesis are organized as follows:

The next chapter provides an comprehensive overview about the state-of-the-art of trace based performance analysis tools. It discusses a number of established software tools for performance measurement and trace analysis including implementation details, trace file formats, internal memory data structures, and evaluation methods. Furthermore, related data compression methods for event traces of different application domains are presented.

The chapter *The Design of the CCG Data Structure* introduces the fundamental design of the newly proposed main memory data structure and the principle of data compression and shows how aspects of lossless and lossy compression methods are combined. For all relevant evaluation and analysis methods, which are required by common analysis and visualization tools, customized alternatives are outlined, that do not require any explicit data decompression.

The fourth chapter *Algorithms for the CCG Data Structure* presents the comprehensive discussion of all algorithms related to the Complete Call Graph data structure. The first part includes the default construction and compression algorithms and their detail components. The second part focuses on the analysis and evaluation algorithms, in particular for navigation and searching in CCGs as well as for statistical summaries and timeline visualization. Besides the essential algorithms, a number of advanced algorithms is presented for the CCG data structure, including the persistent storage, adaptive compression and re-compression, and distributed methods for construction, compression and analysis.

The fifth chapter *Evaluation of CCG Algorithms* contains extensive theoretical and practical validation of the Complete Call Graph approach. At first, it presents a theoretical data compression model and investigates the best and worst case compression behavior with synthetic experiments. Further on, it examines the real-world behavior of CCG compression as well as CCG analysis for *small scale compression* and *large scale compression* which reflect two typical application scenarios. Advanced algorithms for CCG evaluation are covered separately. Finally, recommendations for all major parameters of the CCG algorithms are formulated considering all previous experiment results. A practical performance comparison of the CCG implementation completes this chapter.

The final chapter provides a summary and conclusions of the thesis and gives an outlook on future work.

2 State-of-the-Art in Event-Based Trace Analysis and Related Work

This chapter introduces the state-of-the-art methodology of performance tracing as well as a variety of established tools. Furthermore, it presents trace data file formats and memory data structures for trace data, which is the subject of the presented work. Finally, it provides an overview over related work in the area of compression and reduction of trace data.

An overview on some well known and established tools for interactive and automatic trace analysis forms the introduction to this chapter. Besides a general functionality overview, this will reveal the requirements of those kind of applications. Such software tools are regarded as the main targets for improved memory data structures.

In the following section, several trace file formats are examined as they will provide the data source for the memory data structures. In fact, the existing memory data structures resemble the file formats in some aspects. This means, that not only data is transported from trace files to memory, but also the storage concept is mostly adopted from trace file formats. This could be read as a hint that there has not been much attention to memory data structures for the purpose of program trace analysis. This would also explain the lack of publications about this matter. Nevertheless, there is an entire section dedicated to memory data structures of some well known tools in more detail which has been extracted from parts of publications about the tools as well as source code access and feedback from the respective authors.

The following part of this chapter covers usual types of queries to event trace data structures and their implementation. This serves two purposes, it reveals the usual kinds of queries which need to be supported and it allows an estimation of computational effort and storage requirements. In the end, this will be the basis of comparison to the newly designed approach.

Finally, this chapter will provide an overview of some existing approaches for compression and reduction of event traces. Besides event traces for parallel performance analysis, it also covers memory address traces and MPI replay traces which have partly different properties.

2.1 Tracing and Logging

This thesis proposes a contribution to the analysis of event traces for sequential and parallel performance analysis, in particular in the High Performance Computing (HPC) field. The term *event tracing* expresses the notion of detailed recording of all *steps* (events) in order to reproduce and comprehend the *path* of an algorithm or a program through an abstract space. There is a more general principle called *event logging* which is widely used in computer science and computer engineering. Yet, usually *tracing* and *logging* have a different meaning. First of all, logging is a more coarse grained method. Following the above figure, logging records only landmarks on the path instead of all steps.

Usually, logging mechanisms in software systems record a protocol of important operations, exceptional states, and failure situations. Logging can be used on the operation system level or on the application level. Typical examples are the kernel logs of operation system images or the access log of web servers. The resulting log data is kept for examination at a later time. Furthermore, logging is commonly regarded as a continuous and permanent process. On one hand, this means logging is mostly collected for continuous processes. On the other hand, it is permanently used as a part of the regular operation. Therefore, logging is usually restricted to produce only moderate amounts of data.

Tracing differs from the above conception of logging with respect to the following four characteristics:

detailed: Tracing is more fine grained and produces potentially large results data sets.

limited duration: Tracing is performed during a limited time interval between explicit start and end points. This may be from the start to the end of a program run or only a part thereof.

complete: It captures all events of certain types that occur during the tracing time.

temporary: Tracing is not part of the regular operation of target programs. Instead, tracing is applied temporarily to a pristine program by means of so called instrumentation in order to perform particular investigations. Afterwards, it is removed for the regular "production runs".

For the rest of this thesis, the term *trace* is used in the latter sense.

2.2 Trace Analysis Tools

This section provides an overview about established trace analysis tools and their functionality. It examines internals of state-of-the-art tools for interactive and automatic program trace analysis and allows to anticipate requirements for their data structures and evaluation algorithms.

The tools covered are:

- Vampir and VampirServer
- Paraver and Dimemas
- Jumpshot
- Kojak and Scalasca
- Debugging Wizard (DeWiz)
- Tuning and Analysis Utilities (TAU)

There are many more academic and commercial tools, for example the Intel Trace Analyzer [Cor07], OpenSpeedShop [SGH06], Pajé [dKdOSB00, dKdOSM03], and Paradyn [MCC$^+$95]. All of them share their general approach with one of the above mentionend tools, therefore they are not covered separately.

2.2.1 Vampir and VampirServer

The Vampir tool (*Visualization and Analysis of MPI Resources*) [NAW$^+$95, BNS00, BHNW01] is one of the most well-known and most widely used tools for HPC trace file visualization. It has been available for more than ten years as a commercial product. Like it's predecessor PARvis [NA] it originates from the Research Center Jülich, Germany (formerly named KFA Jülich). Later, the project moved to TU Dresden, where it was continued and extended.

As a complement to Vampir there is the VampirTrace software for code instrumentation and run-time measurement. It is the default tool for generating traces for analysis with Vampir [MKJ$^+$07, KBD$^+$08]. VampirTrace is available as open source software under BSD license since April 2006. It is being developed in collaboration with the Expert trace library by FZ Jülich [WM00a].

Furthermore, there exists a successor version of Vampir called VampirServer which incorporates a newly designed distributed software architecture in order to achieve much higher scalability [BNM03, BMSB03]. This concerns scalability with huge amounts of trace data and scalability with the number of processes or threads [BHNW01, BNM03, BMSB03].

Figure 2.1 shows the distributed components. The client is a local visualization application, which is intended to run on the user's local workstation. It requires only a modest network connection to the server since client and server exchange relatively small amounts of data containing only the display information already adapted to the client's screen resolution.

The server part consists of the master process and number of distributed workers. The master is responsible for data management and communication with the visualization client. The worker processes are assigned disjoint parts of the trace data set. Upon client requests, the master forwards partial requests to the workers. Then, all workers compute partial answers. The master collects the local answers and forwards the final answer to the client.

Parallelization and distributed storage for trace analysis is an obvious way to cope with increasing effort and especially with increasing amounts of trace data in the order of magnitude of tens of gigabytes. The new compression approach of this thesis is an alternative way. In fact, the scalability demands of Vampir and VampirServer were the starting point for this thesis. Therefore, one objective of the new approach will be compliance with the conditions and constraints associated with Vampir and VampirServer. Yet, the distributed storage in VampirServer and the data compression in the CCG approach are complementary. Both have been incorporated successfully in a prototype implementation [KBN05] and might be combined in a future version of VampirServer.

The main displays and the functionality of Vampir and VampirServer are introduced below.

Global Timeline Display

The global timeline display presents an overview of all processes of a parallel program in a space-time diagram similar to a Gantt diagram. The function call behavior is visualized as segments of the process bars. It distinguishes groups of related functions like MPI, user function, I/O, special library calls, tracing overhead etc. by color. Messages, collective MPI communication and input/output operations (I/O) are represented by arrows and lines. Individual arrows are hidden in coarse zoom levels, because they would obscure the whole display otherwise. As soon as a sufficient zoom level is reached, the single message lines are shown.

For both, process bars and arrows detailed context information is provided on demand by clicking on any display item. The horizontal dimension can be zoomend and scrolled arbitrarily to change the time interval and the detail level to be shown. By this means, the whole amount of data can be explored effectively. Otherwise, the user could not cope with all detail information all at the same time, let alone the availability of the corresponding screen or print resolutions. Figure 2.2 shows an example for a typical global timeline diagram, which is based on a trace of the WRF code (Weather Reseach and Forecast Model) [MDG$^+$04].

Process Timeline Display

Local timeline diagrams are very similar to their global counterparts but focus on a single process. The vertical dimension is used to unfold the function call hierarchy of the process, i.e. to arrange the segments of the process bar according to the associated call depth. For comparison of processes several such diagrams can be aligned with synchronized zoom intervals. Figure 2.3 gives an impression of a local timeline for the first process (Id 0) aligned to the global timeline example in Figure 2.2.

As with the global timeline, messages, collective operations and I/O activities are presented with arrows. Again, all components provide further information on demand (by mouse click). Horizontal zooming and scrolling works in the same way as for the global timeline diagram.

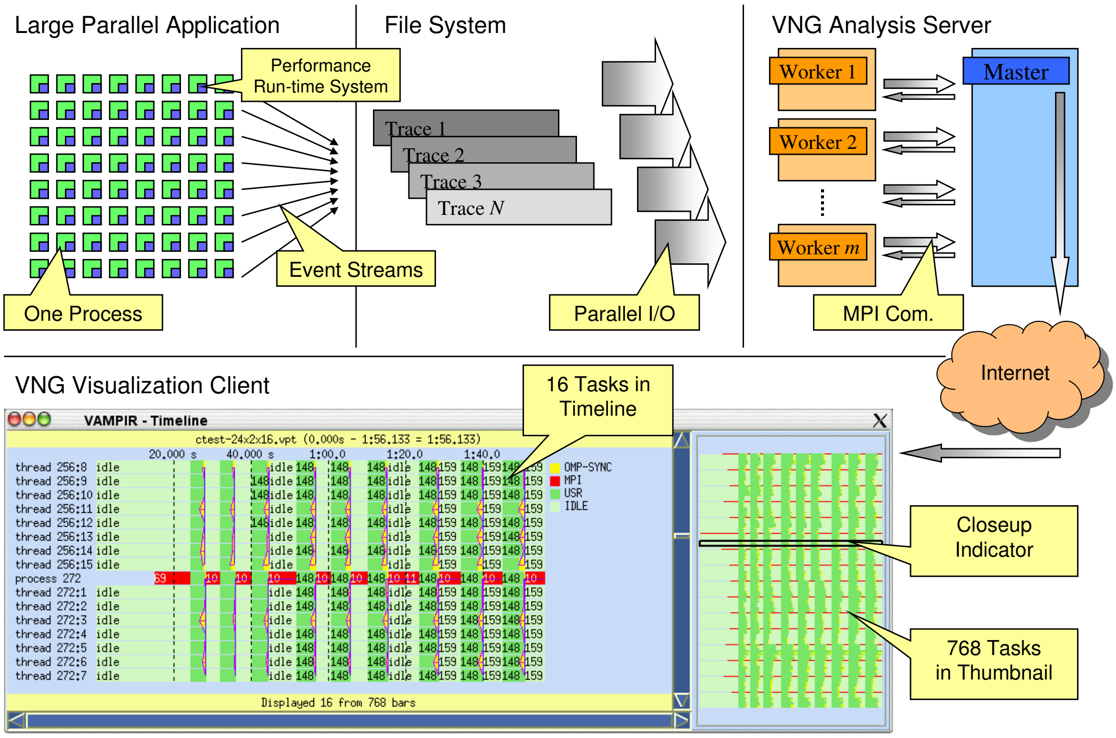


Figure 2.1: Overview of VampirServer's distributed architecture. The measurement system collects local trace information per process/thread (top left) which is collected at a common parallel file system (top middle). The distributed server reads the trace data to main memory in parallel (top right). The display client runs on a remote workstation (bottom). It communicates with the server's master process over a standard internet connection to send requests and receive display data that are already adapted to the pixel resolution of the target windows [KBN05].

Figure 2.2: Vampir global timeline diagram showing 32 of 64 processes of the WRF application. The alternating phases of the parallel program execution are clearly visible. Note that the first process differs from the uniform behavior of all other ones.

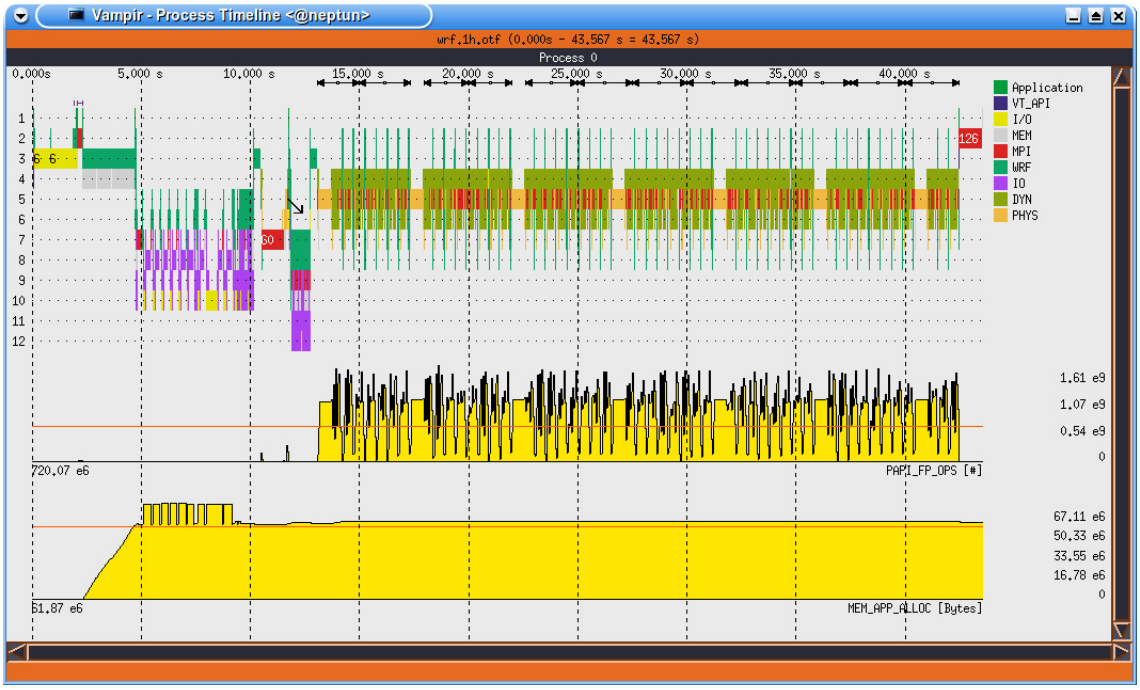


Figure 2.3: Local call timeline diagram showing the master process from the global timeline example in Figure 2.2. The unfolded function call hierarchy allows better insight into the programs procedural structure. This example includes two performance metrics: the floating point rate and the total memory allocation. Both show different behavior over the different phases.

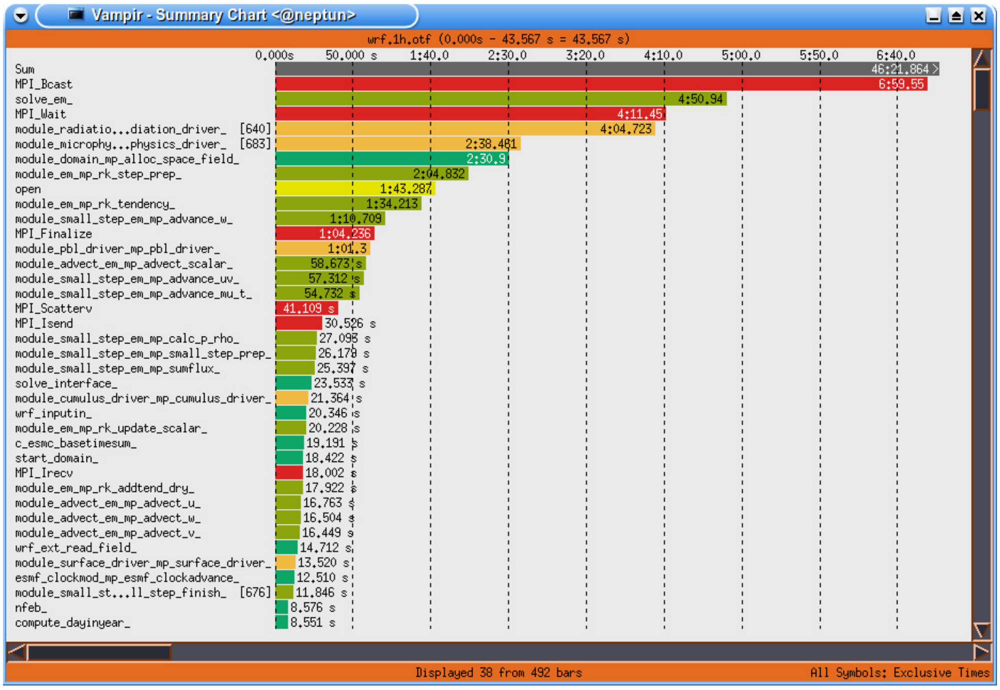
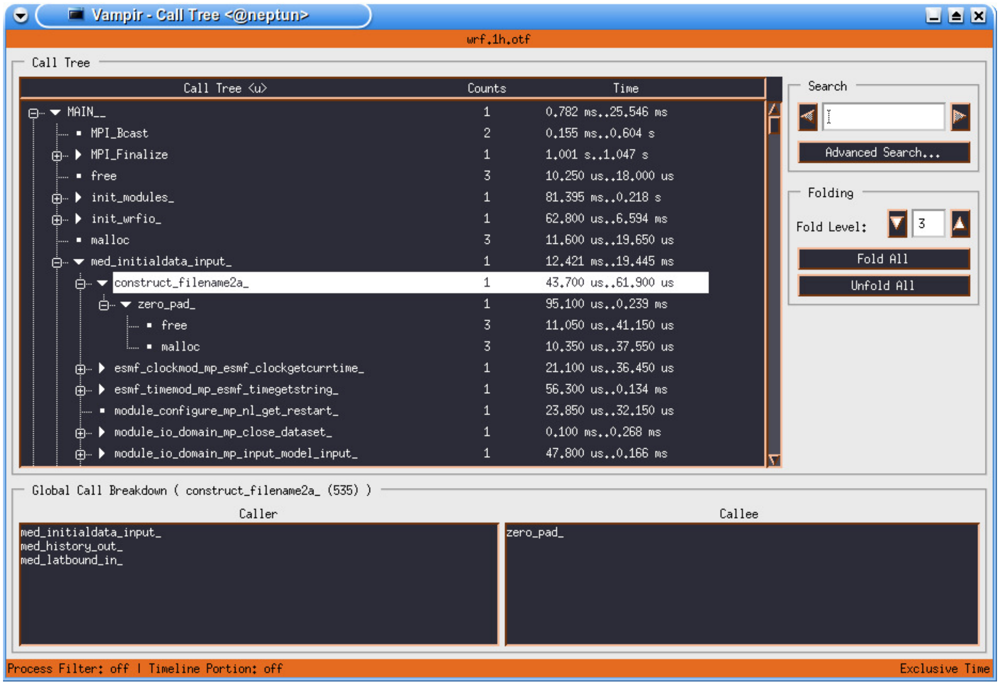


Figure 2.4: Vampir summary chart in bar chart mode. It is also zoomable horizontally to closer examine smaller entries as shown here. All functions are listed separately while the function groups can be identified by color. This display is most useful for detecting the most time consuming functions for a particular time interval.



wrf.1h.otf

Call Tree

Call Tree <u>	Counts	Time
MAIN__	1	0.782 ms..25.546 ms
MPI_Bcast	2	0.155 ms..0.604 s
MPI_Finalize	1	1.001 s..1.047 s
free	3	10.250 us..18.000 us
init_modules_	1	81.395 ms..0.218 s
init_wrfio_	1	62.800 us..6.594 ms
malloc	3	11.600 us..19.650 us
med_initialdata_input_	1	12.421 ms..19.445 ms
construct_filename2a_	1	43.700 us..61.900 us
zero_pad_	1	95.100 us..0.239 ms
free	3	11.050 us..41.150 us
malloc	3	10.350 us..37.550 us
esmf_clockmod_mp_esmf_clockgetcurrtime_	1	21.100 us..36.450 us
esmf_timemod_mp_esmf_timegetstring_	1	56.300 us..0.134 ms
module_configure_mp_nl_get_restart_	1	23.850 us..32.150 us
module_io_domain_mp_close_dataset_	1	0.100 ms..0.268 ms
module_io_domain_mp_input_model_input_	1	47.800 us..0.166 ms

Fold Level: 3

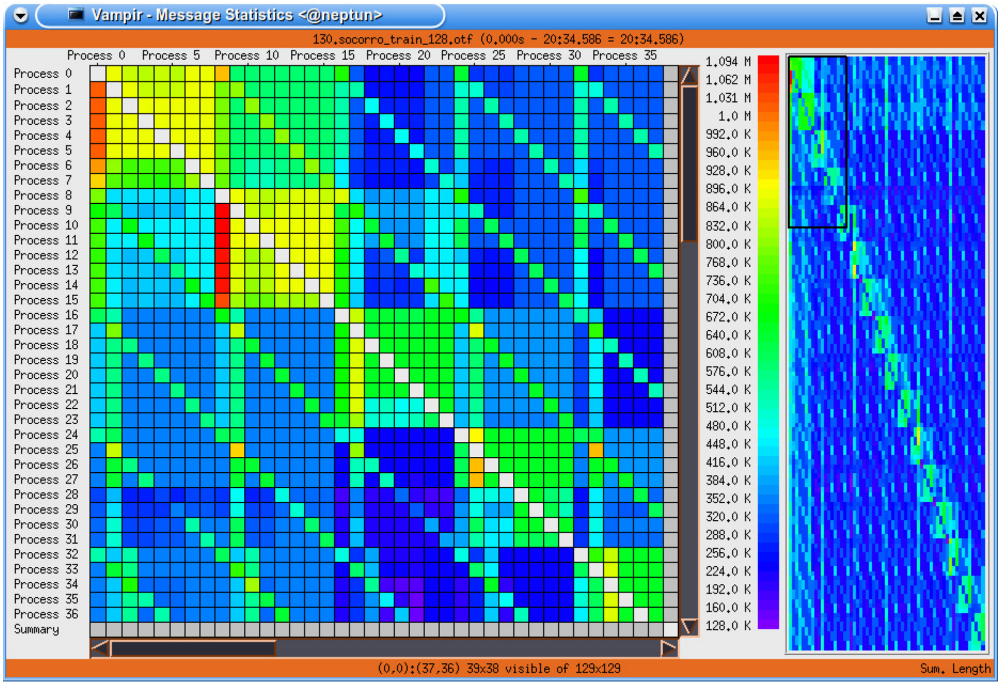
Global Call Breakdown (construct_filename2a_ (535))

Caller	Callee
med_initialdata_input_ med_history_out_ med_latbound_in_	zero_pad_

Process Filter: off | Timeline Portion: off

Exclusive Time

Figure 2.5: Interactive call tree window. The upper right section shows an top-down call tree representation. Below all parent functions (callers) and child functions (callees) to the currently selected functions are listed.



130.socorro_train_128.otf (0.000s - 20:34.586 = 20:34.586)

(0,0):(37,36) 39x38 visible of 129x129

Sum. Length

Figure 2.6: The communication statistics display provides a visual overview about different aspects of inter-process communication with color-coded values. This example shows the total number of bytes transfered from senders (left legend) to receivers (top legend) in a run of the Socorro SPEC MPI benchmark with 128 processes [MKG+04].

In addition, the local timeline diagram optionally shows available performance metrics, for example from hardware performance counters. This allows to watch the the performance metrics aligned to the call hierarchy, providing additional information about individual function calls in an intuitive way. Figure 2.3 shows an example with two performance metrics aligned to the local timeline.

Summary Chart

The summary chart display presents statistics related to function/subroutines or groups of functions. It provides aggregated totals of runtime per function including or excluding child calls as well as number of occurrences. The information can be displayed as absolute values or percentages in linear or logarithmic manner. It is either given globally or for single processes. The user can choose between several representation such as a bar chart, a pie chart or a plain table. See Figure 2.4 for an example.

In principle, this kind of information is identical to the result of profiling. However, it can be computed for arbitrary intervals of time which ordinary profiles cannot. The current time interval is automatically adopted from the timeline diagram. Therefore, the summary charts can be regarded as context information to the current timeline view.

Call Tree Display

The call tree display allows investigation of the relationships of function calls in a trace without respect to temporal order. It provides a top-down tree of all functions with their respective child functions. For exploring child calls as well as parent functions, two additional lists show all parent or child functions of the currently selected function. Again, this is similar to call trees provided by profiling tools but adapts to the currently selected zoom interval. A screenshot of the Vampir call tree display is shown in Figure 2.5.

Communication Statistics Display

The communication statistics display is specifically designed for investigation of the communication behavior between processes. It can provide information about number of messages and the total number of bytes sent or received as well as the communication timing and speed. For all three modes the minimum, average, maximum and sum values can be displayed. The values are visualized in a two-dimensional matrix with color-coded entries as shown in Figure 2.6 with an example of 128 processes from the Socorro SPEC MPI benchmark [MKG$^+$04]. Optionally, only a sub-area of this matrix is displayed because the complete relation of all processes becomes very large for highly parallel traces. Again, all values are computed with respect to the currently selected time interval only.

2.2.2 Paraver and Dimemas

The Paraver and Dimemas tools are developed at the European Center for Parallelism in Barcelona/Spain (CEPBA) since 1991. Paraver is an interactive visualization and analysis tool for event traces not too different from Vampir. Dimemas is a trace based replay and simulation tool, which allows to predict run-time behavior of an existing trace under alternative conditions, e.g. different CPU speed or different communication bandwidth. Furthermore, the so called *DiP* environment contains the common instrumentation and tracing infrastructure. This software is available as a binary package without source code but with a non-commercial license [LGP$^+$96].

The DiP approach supports all common parallel paradigms, namely MPI, OpenMP and hybrid parallel programs. It uses two fixed hierarchical models for processes of parallel programs, as well as ressources of parallel systems [CEP00, CEP01b, CEP01a], see also Section 2.3.7.

Along both hierarchies automatic filtering and aggregation is supported, i.e. events and their properties from one level can be aggregated to be shown in condensed form on the parent level. This allows to provide an overview of performance properties on a high level of the hierarchy, for example the sum of communication volume inside a cluster node or the average floating point rate over several CPUs.

Unlike all other tracing tools, Paraver supports only three very basic event record types [CEP01a]:

- states events, for example function call enter and leave,
- atomic events, for example for performance counter samples, and
- communication events, for example point to point messages or collective communication.

Therefore, it is said to contain semantic-free records [CEP01a]. This means, all specific events are mapped to one of the three record types and specific properties are mapped to generic properties irrespectively of the semantics. So, different properties may be mapped to the same representation in different experiments. When visualizing with Paraver, the generic events are displayed regardlessly of the original meaning. It is left to the user to map the generic display back to the specific semantics. See Figures 2.7 and 2.8 for examples of the Paraver timeline display and the performance counter display.

2.2.3 Kojak and Scalasca

The *Kojak* and *Scalasca* toolkits follow a different approach for performance trace analysis than the previously mentioned tools. Unlike the trace visualization tools, it uses an automatic analysis process to scan a trace for performance critical situations. The result is presented to the user via the special display component called *CUBE*.

The automatic analysis in *Kojak* is performed by the command line tool *Expert* which utilizes the *EARL* library [WM99, Wol04]. Based on *EARL*, the pattern detection tool *Expert* can access and evaluate a trace. The single pattern detection sub-programs are looking for certain pre-defined situations (behavior patterns) inside the event stream. Subsequently, every instance of a detected pattern is classified as critical or noncritical according to a pattern specific severity rating. This allows a graduation from more critical to less critical cases. All predefined behavior patterns are arranged in a hierarchy from general patterns to specific patterns [WM00a], see also Figure 2.9. As an example the so called *late sender* and *late receiver* situations are shown in Figure 2.10. Both concern point to point messages and evaluate the timing of a pair of matching *send* and *receive* events. In this example the severity is calculated as the delay of one call with respect to the begin of the other.

The *Scalasca* toolkit is the successor of *Kojak* that allows to perform automatic analysis of parallel traces in a distributed manner. It includes the successor to *EARL* called *PEARL* (P for parallel) and the successor to *Expert* called *Scout*. *Scalasca's* parallel approach requires the same degree of parallelism as the target trace, i.e. for every trace process there is one analysis process that evaluates all local behavior patterns. For non-local behavior patterns, like point to point messages or collective MPI communication, Scalasca performs a re-play of the original communication. By this means the distributed performance properties are transfered to one communication peer that will perform the evaluation. This scheme is restrictive in the number of parallel analysis processes, yet, it is appropriate for automatic post-processing on the same HPC systems using the same CPUs and ressources as the original run. It achieves an outstanding scalability to tens of thousands of processes [GWWM06, GKP^{+}07].

After the automatic analysis, all results are stored in a single XML output file for visualization by the display component *CUBE*. It uses a custom display design consisting of three categories (also called dimensions, therefore the name CUBE) arranged side by side, see Figure 2.11:

- Performance Metrics
- Call Tree
- System Tree

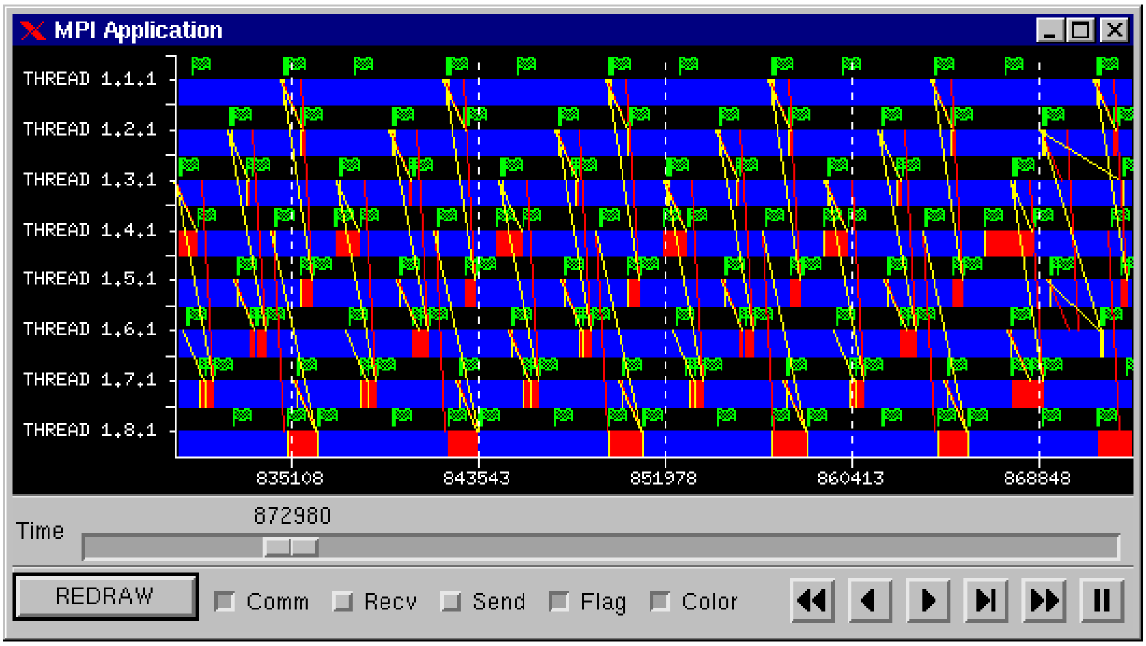


Figure 2.7: Global Paraver display of an MPI program eight processes showing states (function calls) and communication (message arrows). (Taken from [CEP00])

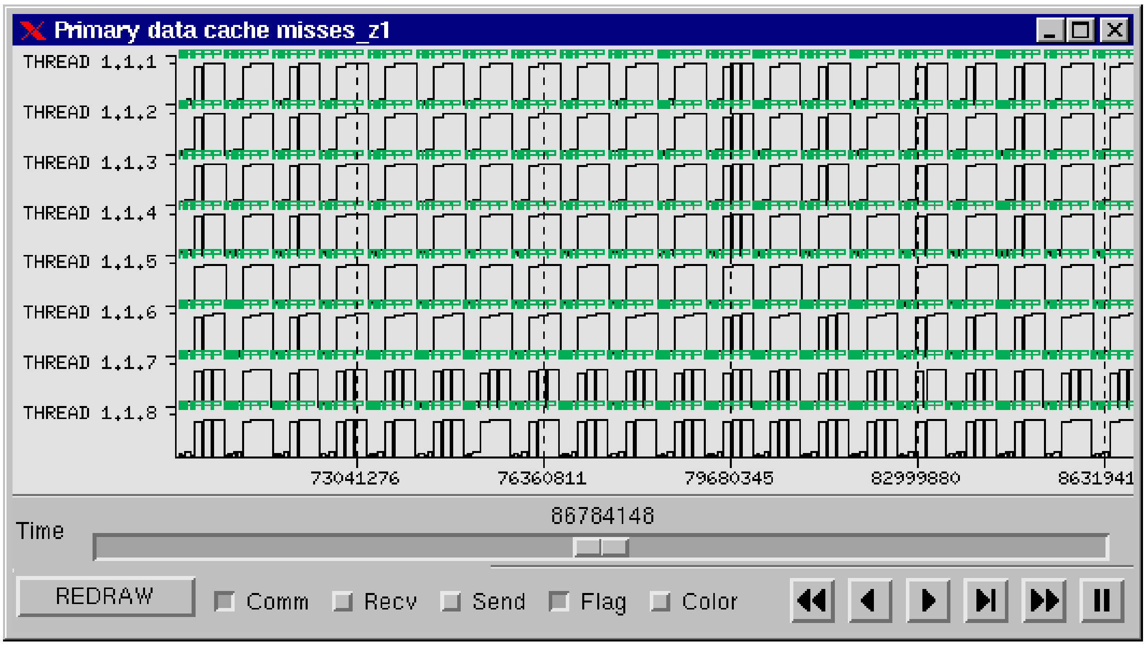


Figure 2.8: Paraver's performance counter display showing data cache misses over time for eight parallel MPI processes. (Taken from [CEP00])

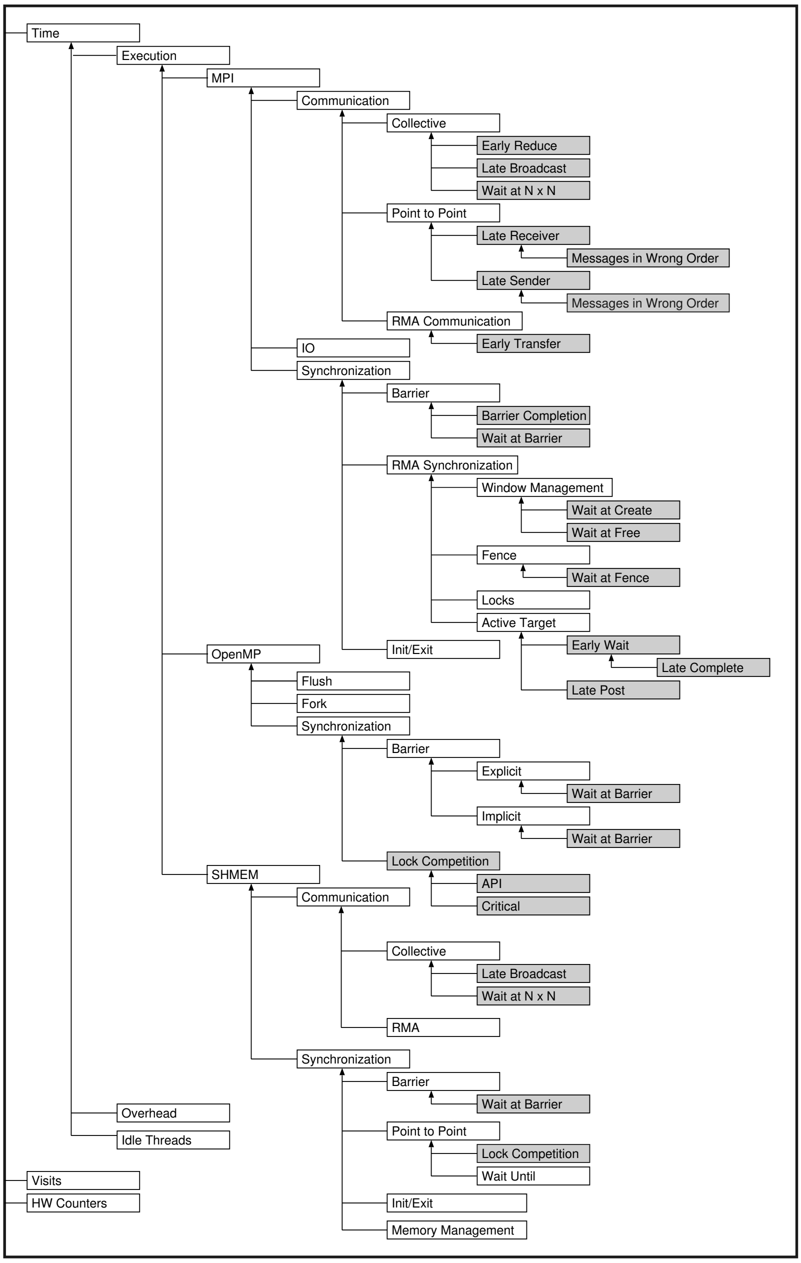


Figure 2.9: Hierarchy of behavior patterns that are evaluated by Expert 3.0. (Taken from [JSC08])

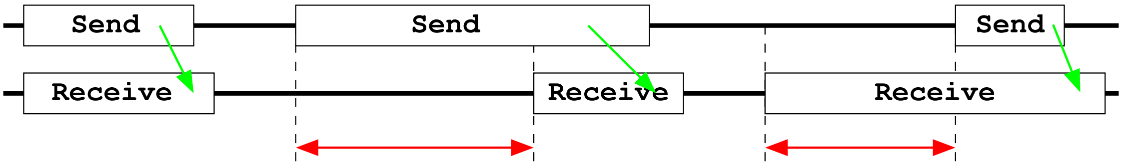
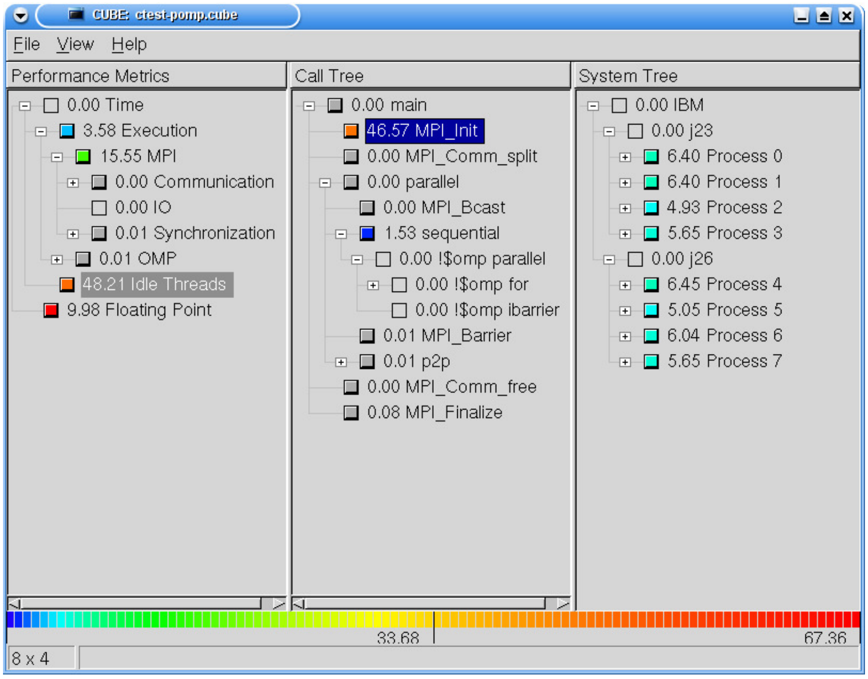


Figure 2.10: Examples of behavior patters for point-to-point messages. In the *late receiver* (middle) and *late sender* (right) situations either the receive or the send operation is delayed.

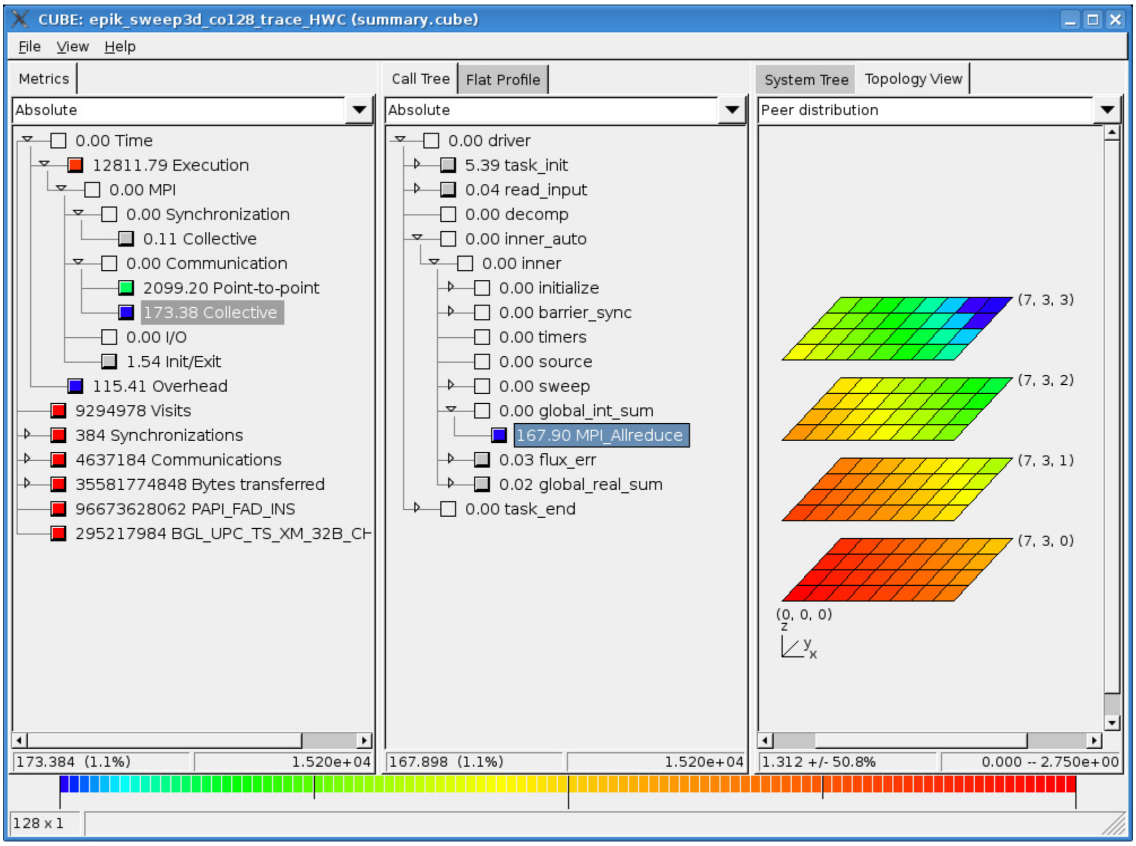


0.00 Time
3.58 Execution
15.55 MPI
0.00 Communication
0.00 IO
0.01 Synchronization
0.01 OMP
48.21 Idle Threads
9.98 Floating Point

0.00 main
46.57 MPI_Init
0.00 MPI_Comm_split
0.00 parallel
0.00 MPI_Bcast
1.53 sequential
0.00 !$omp parallel
0.00 !$omp for
0.00 !$omp ibarrier
0.01 MPI_Barrier
0.01 p2p
0.00 MPI_Comm_free
0.08 MPI_Finalize

0.00 IBM
0.00 j23
6.40 Process 0
6.40 Process 1
4.93 Process 2
5.65 Process 3
0.00 j26
6.45 Process 4
5.05 Process 5
6.04 Process 6
5.65 Process 7

Figure 2.11: The CUBE display with its three categories *Performance Metrics*, *Call Tree* and *System Tree*. This particular example shows that $48.21s$ of run time are wasted by *Idle Threads* (first section) whereof $46.57s$ belong to calls to *MPI_Init* (middle section). This waste of run-time is more or less evenly distributed over all participating processes (last section).



0.00 Time
12811.79 Execution
0.00 MPI
0.00 Synchronization
0.11 Collective
0.00 Communication
2099.20 Point-to-point
173.38 Collective
0.00 I/O
1.54 Init/Exit
115.41 Overhead
9294978 Visits
384 Synchronizations
4637184 Communications
35581774848 Bytes transferred
96673628062 PAPI_FAD_INS
295217984 BGL_UPC_TS_XM_32B_CH

173.384 (1.1%) 1.520e+04

0.00 driver
5.39 task_init
0.04 read_input
0.00 decomp
0.00 inner_auto
0.00 inner
0.00 initialize
0.00 barrier_sync
0.00 timers
0.00 source
0.00 sweep
0.00 global_int_sum
167.90 MPI_Allreduce
0.03 flux_err
0.02 global_real_sum
0.00 task_end

167.898 (1.1%) 1.520e+04

1.312 +/- 50.8% 0.000 -- 2.750e+00

Figure 2.12: The CUBE display with performance properties mapped according to topology information.

The first category shows all behavior patterns reported by EARL. The middle category presents the *Call Tree*, i.e. the source code location of behavior patterns. The third category is the so called *System Tree* which shows the hierarchy of threads, processes, SMP nodes and multiple machines. This gives the physical location of performance issues within the computing system. As an alternative, the last category can show performance properties according to hardware topology information, see Figure 2.12.

All three categories are shown as so called *weighted trees* that change the severity rating associated to single tree nodes upon folding and unfolding: When a node is collapsed, it shows the sum value for the whole sub-tree beneath, whereas an expanded node only shows the exclusive value without contribution of it's children. Besides numeric values (absolute or percentage), the tree nodes are marked by a color coding which allows easy identification of the most important spots in any of the three categories.

For investigation of detailed behavior the three categories are interconnected in a special way: In the beginning, all three categories show the global results. Selecting a sub-category in the left-most tree (performance metrics) will restrict the other categories to the particular property exclusively. Likewise, a selection in the middle category (call tree) makes the last category show details about the selected performance property with regard to the selected part of the call tree. Selecting the root node in any category will undo the restrictions.

The approach of *Kojak* and *Scalasca* is very convenient for detecting the "usual suspects" of performance problems as included in the hierarchy of known problems. Therefore, this is a quick and convenient way to address a number of possible critical performance properties and relieves the user from standard tasks.

2.2.4 Jumpshot

The Jumpshot family from the Mathematics and Computer Science Division at the U.S. Argonne National Laboratory contains several successive trace visualization tools over a long history of over 16 years: Beginning with the Upshot tool in 1991 over the Nupshot tool in 1994 and four major versions of the Jumpshot tools from 1994 to 2007 [HL91, KL94, ZLGS99, WBS$^+$00, CALG07].

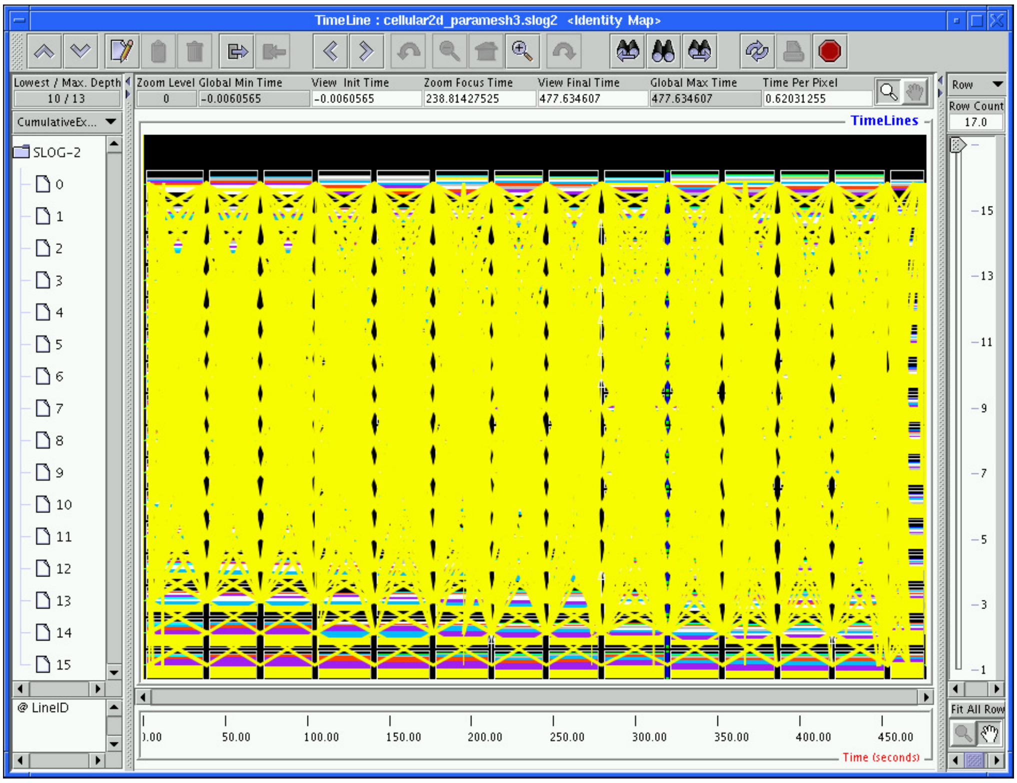
The tools provide timeline and statistics visualization as well as interactive browsing and zooming not unlike the Vampir and Paraver tools, see Figures 2.13 and 2.14 for examples. While the former tools were implemented in the Tcl/Tk script language, the Jumpshot versions 1 to 4 are implemented in Java. The tools were and still are part of the *MPI Parallel Environment* (MPE) which is part of the MPICH implementation of the MPI standard [CGL98].

Unlike other visualization tools, the Jumpshot traces viewers do not load a complete trace file to main memory in order to provide quick interactive response to user events. Instead, only selected parts of the trace data are loaded on demand [CGL00, WBS$^+$00, CALG07]. This provides an advantage in resource consumption when processing very large traces. Yet, at the same time it is a challenge to achieve quick and smooth reload operations for large trace files during interactive visualization.

The Jumpshot tools solve this dilemma by far reaching customization of the trace file formats towards the visualization process. Therefore, all tools are closely coupled to their respective trace formats which evolved alongside, see also Section 2.3.6.

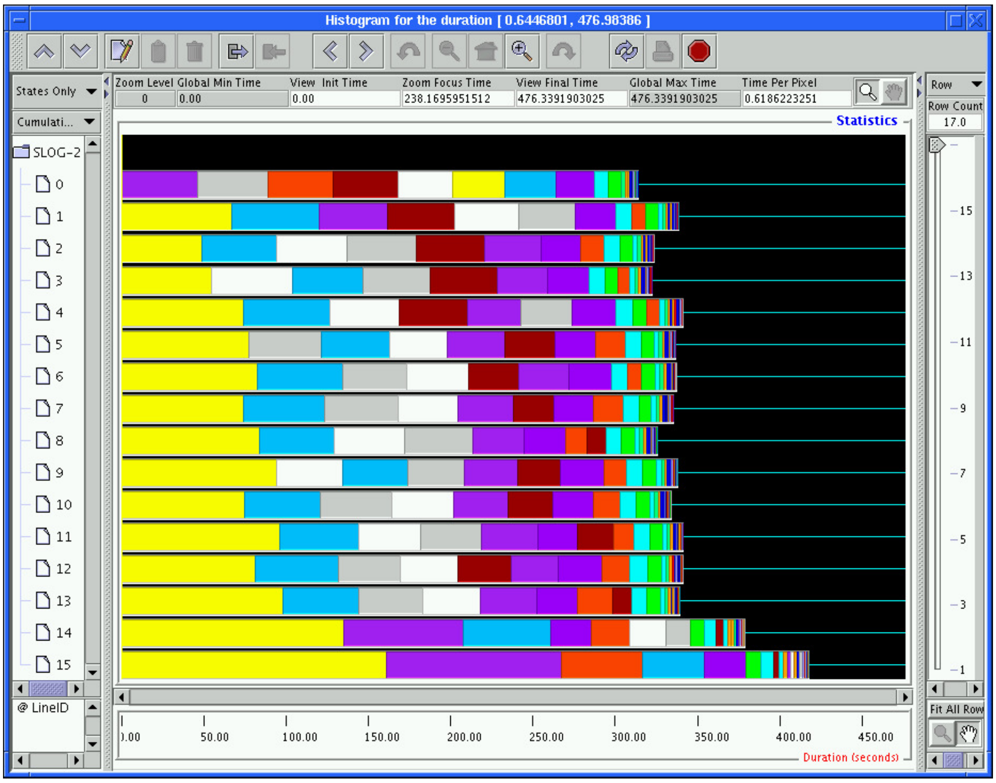
2.2.5 Debugging Wizard (DeWiz)

The Debugging Wizard (DeWiz) is a set of collaborating software components [KSV03, BKN04]. It is developed at the Institute of Graphics and Parallel Processing (GUP) at Johannes Kepler University in Linz/Austria. It is the successor of the tools collection for *Monitoring And Debugging* (MAD) which includes the *EMU* tool (*Event Monitoring Utility*), the debugging tool *ATEMPT* (*A Tool for Event ManiPulaTion*) and many more [KGV96b].


TimeLine : cellular2d_paramesh3.slog2 <Identity Map>

Lowest / Max. Depth: 10 / 13
Zoom Level: 0
Global Min Time: -0.0060565
View Init Time: -0.0060565
Zoom Focus Time: 238.81427525
View Final Time: 477.634607
Global Max Time: 477.634607
Time Per Pixel: 0.62031255
Row Count: 17.0

Figure 2.13: Jumpshot 4 timeline display. (Taken from [CALG07])


Histogram for the duration [0.6446801 , 476.98386]

Zoom Level: 0
Global Min Time: 0.00
View Init Time: 0.00
Zoom Focus Time: 238.1695951512
View Final Time: 476.3391903025
Global Max Time: 476.3391903025
Time Per Pixel: 0.6186223251
Row Count: 17.0

Figure 2.14: Jumpshot 4 statistics display. (Taken from [CALG07])

As the name suggests, DeWiz is mainly regarded as a debugging tool. It primarily focuses on parallel debugging, especially debugging of message passing programs. In the scope of the presented thesis it is interesting, because it also relies on tracing unlike most debugging tools, and furthermore because it contains an innovative approach with respect to storage of trace information.

DeWiz is a modular framework of encapsulated software components that communicate by means of a standard interface. There are various components available responsible for measurement, data collection and storage, automatic analysis and visualization. Actual DeWiz tools are created by dynamically connecting a number of components for input, processing, graphical output etc. This can be done by setup scripts or manually with an interactive graphical tool. Single components can be implemented in Java or C/C++. There may be multiple components for particular tasks, for example the communication between the components can be chosen to use different communication modules, e.g. for shared memory or plain TCP/IP connections. This is especially important to reduce measurement overhead during online analysis which is explicitly supported by DeWiz [KSV03, BKN04].

One of the major aspects of DeWiz is debugging of message passing behavior including pattern detection in communication and non-deterministic behavior. This particular and special part is discussed for the rest of this section [Kra02, KKN04]. The MPI programming model [For95, For97] explicitly allows wildcard parameters in communication routines. This induces the problem of *race conditions* which is generally found in parallel programming. Program errors caused by race conditions are particularly hard to discover by standard debugging techniques because they appear sporadically and are not reliably reproducible [Kra00].

Event Graph

DeWiz approaches error detection with the *Event Graph* model for causality relationships between events [Kra00]. The Event Graph relies on the *happened before* relation [Lam78] which specifies the causal dependency between events. It is defined as follows:

Definition 1. *Let* $\{e_p^i; i \in I, p \in P\}$ *be a set of events with a sequence number* $j \in N$ *on processes* $p \in P$*. The happened before relation* $\rightarrow$ *is the smallest transitive and irreflexive relation* $\rightarrow$ *that satisfies the following two conditions:*

1. *If events* e_p^i *and* e_p^j *happen on the same process* p *and* e_p^i *occurs before* e_p^j*, i.e.* $i < j$*, then* $e_p^i \rightarrow e_p^j$*.*
2. *If event* e_p^i *sends a message from process* p *which is received by event* e_q^j *on process* q*, then* $e_p^i \rightarrow e_q^j$*.*

This relation establishes a partial ordering. Events e_p^i and e_q^j with $e_p^i \nrightarrow e_q^j \wedge e_q^j \nrightarrow e_p^i$ are called *concurrent* $e_p^i \parallel e_q^j$. In particular, this definition states $e_p^i \nrightarrow e_p^i$. It follows $e_p^i \parallel e_p^i$ which is desirable.

User Interface

This Event Graph is presented to the user as a space-time diagram, for examples see Figures 2.15, 2.16 and 2.17. Some events and messages are marked with special colors to highlight error situations, e.g. invalid parameters, mismatching sender-receiver pairs, etc. Additional information for events can be requested interactively, for example explicit message properties, source code information or the set of all logical predecessor and successor events, i.e. all events affecting the current one or affected by it.

Last but not least, race condition situations are marked. Besides observation of race conditions, DeWiz also allows modifying the event graph in order to change the outcome of a race situation. This allows to study an alternative situation which could have appeared instead of the actual outcome. By this means, it is possible to investigate potential errors due to race conditions not (yet) encountered. Furthermore,

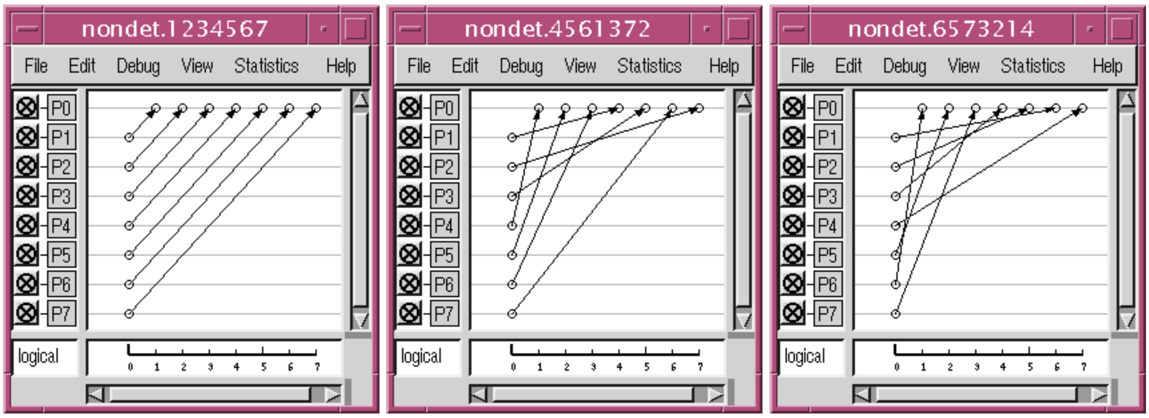


Figure 2.15: Dewiz display of alternative outcomes of a wildcard communication. (Taken from [Kra00])

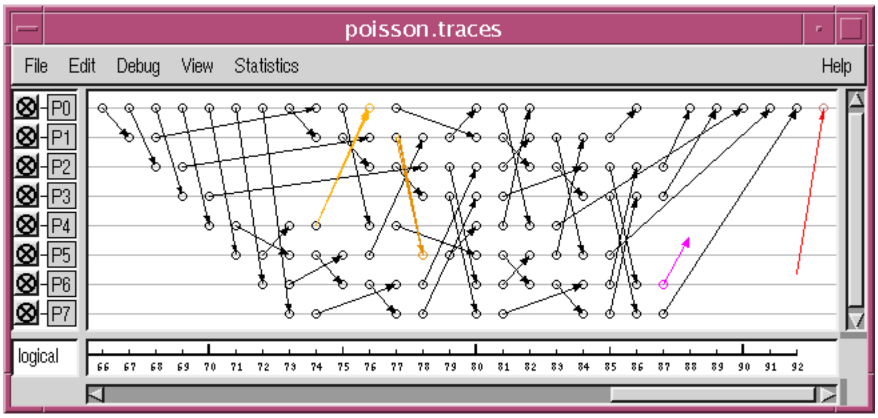


Figure 2.16: Indication of special message properties: Conflicting properties in send and receive events (orange) and isolated send and receive events (purple and red). (Taken from [Kra00])

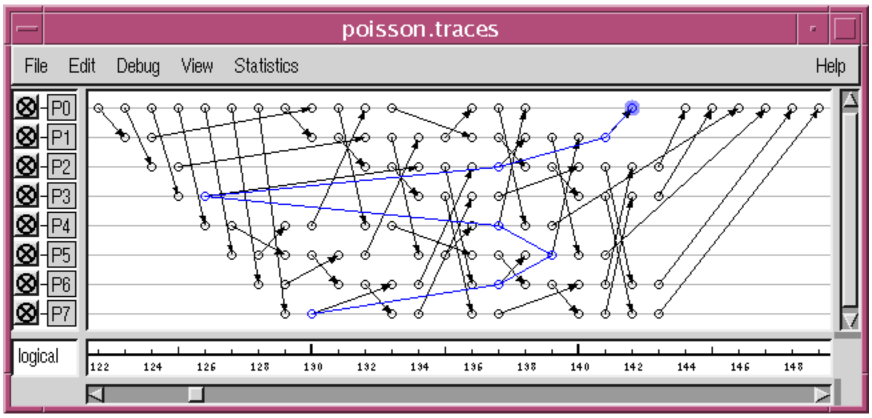


Figure 2.17: Coordinated parallel break point to a local break point in process P0. (Taken from [Kra00])

DeWiz supports propagating the unmodified or modified event graph to a special MPI run-time system for guided execution: That means the following execution of the target application (with the same parameters and the same input) is forced to show the specified outcome of the race conditions [KGV96a]. This record and replay scheme [Kra00] is most useful in order to make parallel debugging runs reproducible. This special aspect is ignored by most other parallel debugging tools.

2.2.6 Tuning and Analysis Utilities (TAU)

The Tuning and Analysis Utilities (TAU) constitute a joint effort of the Performance Research Lab [UOR] at the University of Oregon/USA, of the Advanced Computing Laboratory [LAN] of the US-American Los Alamos National Laboratory (LANL) and of the Central Institute for Applied Mathematics (ZAM) [ZAM] at the Research Center Jülich/Germany. It aims to be a portable toolkit for profiling and tracing of parallel programs covering instrumentation, measurement, data recording, event selection and filtering as well as actual performance analysis. TAU provides interfaces to Fortran, C, C++, Java and Python.

The TAU infrastructure for data acquisition and pre-processing is very flexible and portable and explicitly supports third party analysis tools [SM05]. On behalf of actual performance analysis, TAU uses a profiling approach, i.e. it does not look at individual events but at summarized information. Through this means, TAU achieves outstanding scalability and low overhead even for parallel applications with tens of thousands of processes [MWD$^+$05].

TAU's profile visualization tool *ParaProf* provides a variety of different displays visualizing various kinds of profile data in different ways. This includes exclusive and inclusive aggregated run-times and a number of occurrences for all instrumented functions. For examples see Figures 2.18, 2.19 and 2.20. Furthermore, the caller-callee-relations of functions can be investigated with ParaProf, either as plain graph visualization or accompanied with selected statistical properties, see Figure 2.19 for examples.

Besides two-dimensional diagrams, ParaProf offers a three-dimensional display which is able to present interdependencies of up to four items. This is achieved by mapping three properties to the spatial dimensions and an additional one to the color axis. Figure 2.20 gives an impression thereof.

Another important aspect of profile analysis is comparative analysis of multiple profiles. This is very convenient for many purposes. At first, for evaluation of scalability using profiles of an application with different degrees of parallelism, but also for comparing different program versions, for example before and after an optimization. Furthermore, it is interesting investigate the behavior of the same program on different platforms. Last but not least, related properties from the same profile can be compared in order to study correlation effects.

Phase-Based Profiling

Program behavior that is changing over the course of time is an important issue for performance analysis. However, this is not covered by traditional *flat profiling* at all. Therefore, TAU incorporates so called *phase based profiling* to address this [MSM05]. *Phase profiles* can be regarded as a composition of multiple traditional profiles for disjoint *phases* of a program run. Those phases are to be defined explicitly – either statically at compile time or dynamically during program execution. Furthermore, phases can be nested within one another but must not overlap, i.e. they have a FIFO property like program function calls (see also Lemma 2 ins Section 4.1.1).

Phase based profiling can capture the dynamic behavior with a certain granularity. Provided there is a sufficient set of phases, changing program behavior can be tracked by varying statistical properties. Still, phase based profiling preserves the economical storage requirements of traditional profiling. This is one of the fundamental advantages of profiling, especially in contrast to all tracing approaches.

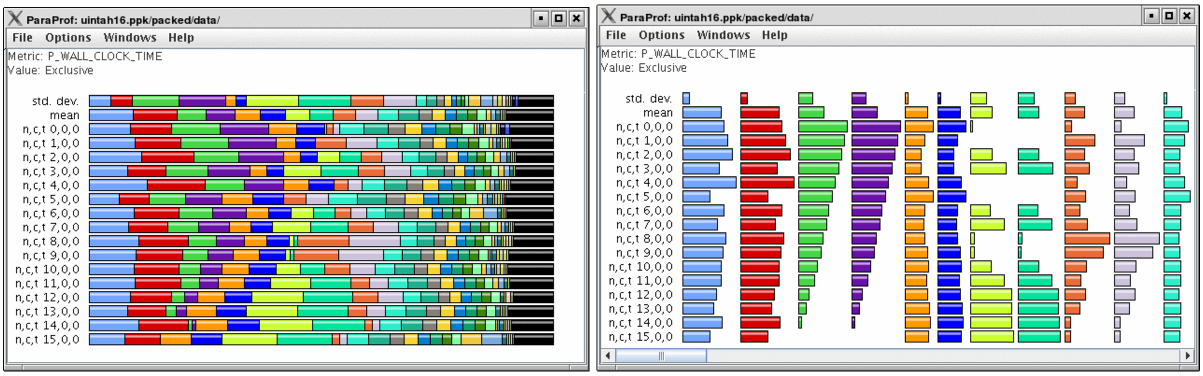
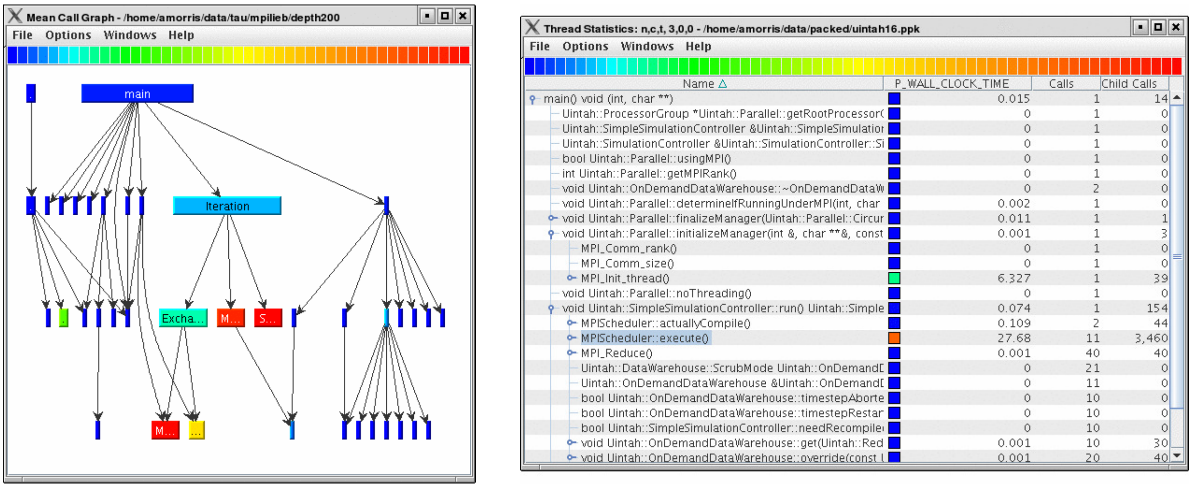
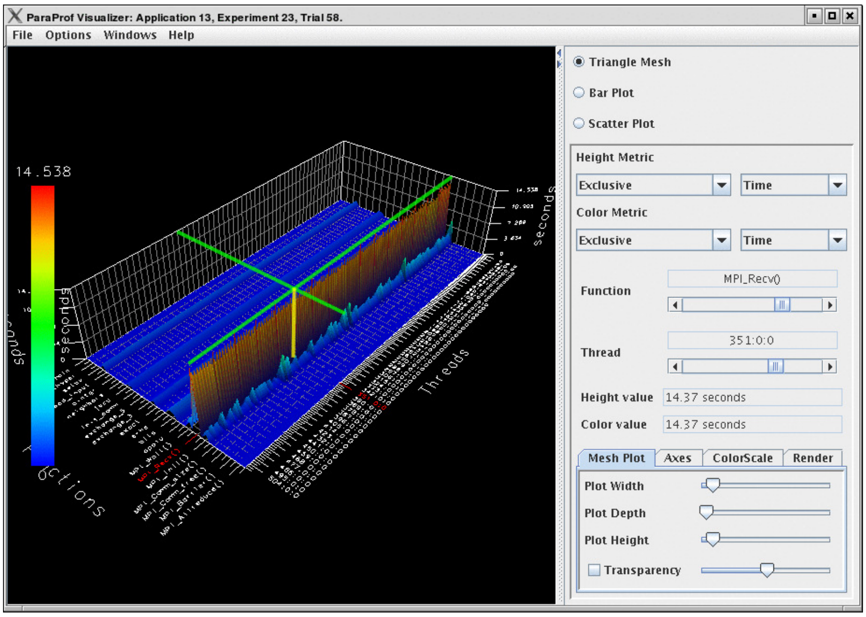


Figure 2.18: ParaProf showing aggregated exclusive run-time per function (colors) per process (rows) in stacked mode (left hand side) or non-stacked mode (right hand side).



Name △		P_WALL_CLOCK_TIME	Calls	Child Calls
main() void (int, char **)	🟦	0.015	1	14
Uintah::ProcessorGroup *Uintah::Parallel::getRootProcessor(	🟦	0	1	0
Uintah::SimpleSimulationController &Uintah::SimpleSimulatior	🟦	0	1	0
Uintah::SimulationController &Uintah::SimulationController::Si	🟦	0	1	0
bool Uintah::Parallel::usingMPI()	🟦	0	1	0
int Uintah::Parallel::getMPIRank()	🟦	0	1	0
void Uintah::OnDemandDataWarehouse::~OnDemandDataW	🟦	0	2	0
void Uintah::Parallel::determineIfRunningUnderMPI(int, char	🟦	0.002	1	0
void Uintah::Parallel::finalizeManager(Uintah::Parallel::Circur	🟦	0.011	1	1
void Uintah::Parallel::initializeManager(int &, char **&, const	🟦	0.001	1	3
MPI_Comm_rank()	🟦	0	1	0
MPI_Comm_size()	🟦	0	1	0
MPI_Init_thread()	🟩	6.327	1	39
void Uintah::Parallel::noThreading()	🟦	0	1	0
void Uintah::SimpleSimulationController::run() Uintah::Simple	🟦	0.074	1	154
MPIScheduler::actuallyCompile()	🟦	0.109	2	44
MPIScheduler::execute()	🟧	27.68	11	3,460
MPI_Reduce()	🟦	0.001	40	40
Uintah::DataWarehouse::ScrubMode Uintah::OnDemandI	🟦	0	21	0
Uintah::OnDemandDataWarehouse &Uintah::OnDemandI	🟦	0	11	0
bool Uintah::OnDemandDataWarehouse::timestepAborte	🟦	0	10	0
bool Uintah::OnDemandDataWarehouse::timestepRestar	🟦	0	10	0
bool Uintah::SimpleSimulationController::needRecompile	🟦	0	10	0
void Uintah::OnDemandDataWarehouse::get(Uintah::Red	🟦	0.001	10	30
void Uintah::OnDemandDataWarehouse::override(const I	🟦	0.001	20	40

Figure 2.19: ParaProf's call graph display (left hand side) and call tree display (right hand side).



● Triangle Mesh
○ Bar Plot
○ Scatter Plot

Height Metric: Exclusive, Time

Color Metric: Exclusive, Time

Function: MPI_Recv()

Thread: 351:0:0

Height value: 14.37 seconds

Color value: 14.37 seconds

☐ Transparency

Figure 2.20: Three-dimensional profile view visualizing exclusive run-time per function per thread.

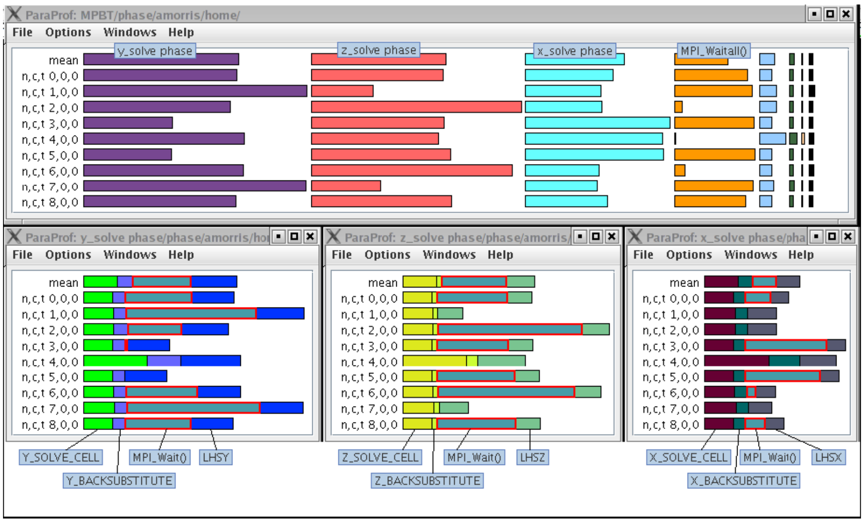


Figure 2.21: ParaProf's two-level displays for phase-based profiles. In the top window the three phases are shown in a compound way. The phases correspond to three main functions of an iterative solver software [BHdW$^+$95]. Some operations appearing outside of explicit phases can be shown separately. The three phase-specific displays below present the details for the phases separately for all processes. Smart highlighting allows the identification of common function calls over all phases. (Taken from [MSM05].)

The definition of phases is not automated but requires user intervention, yet it allows to attach a semantic meaning to every phase. Phases can be defined by structural, logical or execution time aspects of a program. Important sub-routines or library calls may define special phases. Also, stages of an underlying algorithm may induce phases, e.g. initialization, finalization and iteration. Finally, special conditions during execution might be considered as separate phases, e.g. error recovery, see also [MSM05].

ParaProf provides a special two-level display mode for phase-based profiles. On the more abstract level, every phase is displayed in summarized mode as overview. On the more specific level, all items of a single phase are shown separately. See Figure 2.21 for an example with three distinct phases.

Besides profile data, TAU also supports native tracing. TAU's measurement infrastructure is capable of generating sets of profiles as well as traces. There is also an own TAU trace format along with tools to extract TAU profiles from various event trace formats. The former is studied in Section 2.3.9, the latter will be subject to more elaborate discussion in Section 2.4.4.

2.3 Trace File Formats

This section presents details about a number of trace file formats that are connected to the previously introduced tools. It will underline the common basic design of all trace formats and point out individual special solutions.

2.3.1 Common Design of Trace File Formats

All trace file formats share a basic design and have many similarities. First of all, they store the trace information in a (more or less) plain and unprocessed form. Furthermore, trace events are represented by so called *trace records* which cover one event at a time and mark the smallest units of information in all formats. The record types can be grouped into two categories:

- event record types and
- definition record types.

Usually, there are separate record types for all of the different event types (see below).

Event Records

Event record types describe actual events, i.e. distinguished incidents during execution. All *event record types* contain at least a *time stamp* which specifies the point in time when the event happened. Furthermore, most event records provide a location specification. This is either a logical location with respect to the structure of the application (like a thread or process ID) or a physical location with respect to the computing system (like machine, computing node, CPU, core, etc.). Besides time and location, there are additional event properties which are specific for certain types of events.

Event records account for the vast majority of records in large traces. Therefore, efficient encoding of event records is crucial as it directly influences total trace size. Event records are arranged in a sequential stream or in multiple parallel streams. Always, the events are sorted according to their original temporal order.

The most important event record types supported by all major trace formats in a very similar way are:

- *enter* and *leave* of functions/regions,
- *send* and *receive* of point-to-point messages,
- collective MPI communication operations, and
- hardware performance counter samples.

Further examples are:

- I/O activities,
- mutex locking and release, and
- begin and end of OpenMP parallel regions.

Definition Records

Besides event records all formats support a number of so called *definition record types*. Those records provide various global properties necessary for later analysis and for convenience. This includes for example the timer resolution for all time stamps, date and time information of trace creation and platform information like host name, processor type, processor speed, memory size, etc.

The special definition record types are essential in order to increase the over-all storage efficiency, in particular for labels, names or descriptions like function names, names of computing nodes, source file paths etc. Such names come as arbitrary long text and may be referenced very often. In order to increase encoding efficiency, those labels are mapped to integer tokens (identifiers) of fixed size by definition records. The tokens are used whenever a reference to the label would be required in event records or in other definition records. This saves storage space for multiple references.

Common examples for token definition records are:

- process definitions,
- process group definitions,
- function definitions,
- function group definitions,
- source code location definitions, and
- performance counter definition.

In contrast to event records, definition records are usually not critical with respect to their number and their storage size. Even if there are very many definitions the number of events referencing those definitions is usually much higher.

Library Interfaces

For all trace formats there are support libraries which provide interfaces for read and write access to the trace files, such that encoding and parsing issues can be hidden from the programmer and platform issues like variable type sizes or endianness can be resolved transparently.

All trace format libraries support the standard access scheme where events have to be read and written in temporal order. Usually, the libraries provide a set of methods for writing of the individual record types. For reading, the trace events are delivered to the consumer via call back handlers that have to be registered with the library beforehand. Again, there are separate handlers for the different record types.

In addition to this, some trace format libraries provide more or less efficient selective read access, either concerning certain record types or concerning selected processes in parallel traces or concerning selected time intervals. Furthermore, all trace formats include support tools which are in particular necessary for gathering multiple process traces into a single parallel trace.

2.3.2 The Vampir Trace Format Version 3 (VTF3)

Vampir Trace Format 3 (VTF3) is directly connected with Vampir and VampirTrace. Its predecessor VTF has been developed by FZ Jülich like Vampir and VampirTrace. The later VTF3 has been developed at the Center for High Performance Computing (ZHR) of TU Dresden. After the first version of Vampir Trace Format there was no distinct VTF2 though, but evolving intermediate versions.

VTF3 features three equivalent sub-formats with different encodings, called the *binary*, the *ASCII* and the *Fast ASCII* sub-formats. The former has advantages with respect to storage space because of the more dense encoding. All subformats, however, allow ZLib compression on top of them [lGA02]. The two latter sub-formats allow to manually read or modify traces with standard text tools for debugging and testing purposes. The difference between both is a more elaborate encoding with verbose keywords vs. a very terse encoding.

The VTF3 format contains all usual record types as well as some special purpose records, which are partly experimental or deprecated. It collects all records into a single file. All definition records have to be placed at the beginning followed by the sorted stream of event records. Therefore, the VTF3 access library allows only strictly sequential read-through.

The tool *vptmerge* accompanies VTF3. As the name suggests, it merges multiple process traces into a single parallel trace. Furthermore, it is also capable of sorting traces with non-monotonic timestamps. Sorting huge traces is not limited by main memory size but only by disk space.

2.3.3 The Structured Trace Format (STF)

The Structured Trace Format [STF07] has been developed by Pallas GmbH in cooperation with the Center for High Performance Computing (ZHR), TU Dresden in 2001. After the acquisition of Pallas GmbH by the Intel corporation [Cor06], the availability of STF is limited to Intel platforms. The current version of the STF format is part of the *Intel Trace Collector* and *Intel Trace Analyzer* tools.

STF's design goals explicitly names enhanced scalability for large trace file sizes and a very large number of trace processes. Therefore, it features a storage scheme of multiple files per trace with adjustable granularity. In particular, there are several types of files for a single trace:

- a global index file referencing all other files,
- a global declarations file containing definition records,
- a so called *frames* file which contains summary information and special *thumbnail* information,

- a statistics file with statistics about event record types,
- n event data files containing the actual event records,
- n anchor files with index and history information for the event data files,
- a single file for all point-to-point message records, and
- a single file for collective communication records.

STF is able to distribute m trace processes to $n \leq m$ files with a given number of $\lceil m/n \rceil$ processes per event data file. When accessing only a sub-set of available trace processes, the STF library will touch only event data files concerned by the very request. By this means, efficient parallel access is provided as parallel tasks can read disjoint parts of a trace. Yet, the existence of singular files for point-to-point messages and collective communication impairs efficiency and limits scalability because those two files are always to be included.

Another fundamental problem is caused by the separate placement of communication records: When two events in the same process trace are mapped to the same timestamp (e.g. due to limited timer resolution) then their original order is preserved by the order of the event records in the file. As soon as those records are placed in different files, the order is ambiguous – it cannot be reliably restored during merging.

Besides parallel access, STF also provides advanced selective access with respect to so called *frames*. In order to achieve this efficiently, STF relies on supplementary anchor information for every event data file. Those contain a set of explicitly stored file positions where it is safe to start reading. The anchors are accompanied by so called *history* information which provides the full state of the trace processes at a given time stamp. With this additional information it is useful to start reading at this position.

Another particular feature of STF is the *thumbnail* information for the *frames*. It provides a statistical overview about the contents of the frames. Thus, it is possible to look at the concise thumbnails first, and to select the frames to load thereupon. Unfortunately, the encoding of STF is purely binary and not publicly documented. There are neither publications available about record representation nor about I/O performance and scalability.

2.3.4 The Open Trace Format (OTF)

The Open Trace Format (OTF) [KBB$^+$06] is actively developed by the Center for Information Services and High Performance Computing (ZIH) at TU Dresden in corporation with the Performance Research Lab at the University of Oregon. It has been funded by the Lawrence Livermore National Laboratory (LLNL) of the University of California. The Open Trace Format has been designed as successor to VTF3 and STF as a scalable and free parallel trace format and is available under the BSD open source license.

OTF organizes trace records in multiple *streams*, i.e. separate fragments that can be accessed independently (see Figure 2.22), by which selective access and parallel I/O can be achieved. A *stream* contains events of one or more processes in one file in temporal order while every trace process is mapped to one stream exclusively.

In addition, there is an index file that stores the mapping of processes to streams. Figure 2.22 provides an impression of OTF's storage scheme. The process-to-stream mapping can either be specified explicitly or created automatically when a trace is written via the OTF library. When reading event records via the OTF library, the multiple streams are handled completely transparent: The user can select which processes to read, then the OTF library accesses only streams containing those processes. Multiple streams are merged on-the-fly in order to deliver a single sorted stream of events. The number of streams is not limited by the number of file handles available.

The internal representation of records in OTF uses a platform independent ASCII encoding which allows to find resumption points for reading at arbitrary file positions. Based on this, search for time stamps in a sorted stream is achieved by binary search on a file, by which very efficient selective access is supported

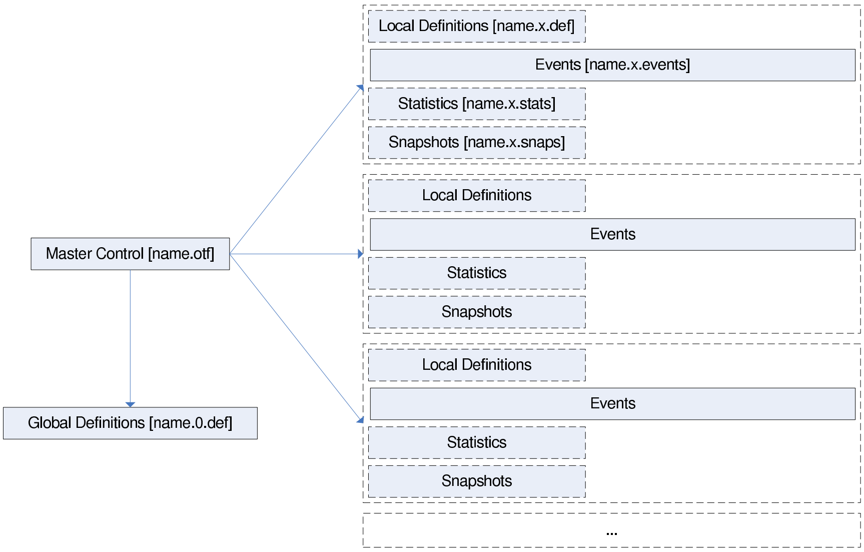


Figure 2.22: The OTF storage scheme: the index file, the global definitions file and the event files are mandatory, all local definitions, snapshots and statistics are optional. (taken from [KBB08])

with $O(\log n)$ effort [KBB08]. For the sake of efficient storage size, OTF supports transparent ZLib data compression [KBB08, lGA02]. In order to support selective access for compressed files a blockwise scheme is implemented. It decreases storage volume to 20% - 35% of the original size on average, compensating for the ASCII encoding.

Definition records in OTF can either be declared global or local, i.e. for the whole trace or only for a particular stream. Furthermore, there are two more classes of auxiliary records besides definitions and events [KBB08]. There are the so called *snapshot records* of different types which provide resumption points for reading. A set of snapshot records allows to re-create the complete state of a trace process at a certain time stamp. Therefore, it is not necessary to read a stream from the beginning.

The so called *summary records* allow an overview about the behavior of a trace process over certain time intervals. They provide statistics about various properties which is useful in order to determine the processes and time intervals to examine in detail by means of selective access. All snapshot and summary record types are derived from corresponding event record types. Both classes of auxiliary record types can be generated from the complete set of event records by the support tool `otfaux`. All auxiliary records are optional and reside in separate files per stream, see Figure 2.22. This allows to generate, re-create or remove them without affecting the event streams.

Finally, the OTF library supports reading and writing of events, definitions, snapshots and summaries with two interfaces: the high level interface for accessing whole traces and the low level interface for reading and writing from/to single streams. The former is useful for reading complete traces during analysis, while the latter is convenient for measurement systems that need to write parallel streams in an independent manner during run-time.

2.3.5 The Epilog Trace Format

The Epilog trace format (*Event Processing, Investigating and Logging*) [WM04] has been developed by Forschungszentrum Jülich GmbH, Germany and University of Tennessee, Knoxville, USA. It is part of the Kojak project and is available as open source software under the BSD license [koj05]. It is fully documented in a detailed technical report [WM04].

The Epilog format uses a binary encoding. It supports all the usual definition and event record types, see Section 2.3.1. There are special record types for tracing events related to OpenMP multi-threading [CJP07]. Epilog also uses a fixed location specification for the physical and logical placement of events with the 4-tuple $(machine, node, process, thread)$.

Another particular feature is Epilog's support for explicit time synchronization. If computing nodes or processes use different local timers, then a synchronization record can specify the local-to-global time stamp transformation at certain points during tracing. Based on such information, it is possible to perform a global time adjustment as a post-processing step [Rab00, WM04]. Last but not least, Epilog has special record types to mark certain events induced by tracing itself, for example de-activating and re-activating of tracing or occasional I/O caused by the tracing sub-system.

2.3.6 The Jumpshot Trace Formats

Associated to the successive visualization tools from the Jumpshot family, as presented in Section 2.2.4, there is a number of trace formats. Unlike all other tools, Jumpshot uses the terminology *event logging* instead of *event tracing*.

The various trace formats evolved together with the successive visualization tools. All of them use binary encoding and consist of a single file. First, there were the *ALOG* format and its successor *BLOG* [HL91, KL94]. They utilize a fixed record encoding consisting of six integer values plus a 12 character text string. The following *CLOG* format eliminated the fixed length encoding in order to allow more flexible extensions and additions of record types [ZLGS99]. Those three formats are typical *event based formats* in the Jumpshot terminology, i.e. they use trace events as the smallest entities of data.

The *SLOG-1* format marks the transition to a so called *state based format*. Now, the smallest data items are states, which have a beginning and an end with associated time stamps. A state corresponds to two *enter* and *leave* events in an *event based format* with two time stamps. The states are organized in so called *frames* which contain all states of consecutive disjoint time intervals. Such frames can be loaded separately, in order to support loading on demand, see Section 2.2.4. In particular, this scheme requires to duplicate bordering states that are shared between adjacent frames in a post-processing step [WBS$^+$00]. This is a remarkable feature for a trace format: It is no longer a more or less passive container for pristine measurement data. Instead, it becomes an active part of the visualization tool and modifies the contents for this purpose.

The latest *SLOG-2* format goes even further in the direction of *SLOG-1*. It is a so called *drawable-based format* which means that now drawing objects are used as smallest entities. Instead of the previous frame structure, the records (drawable objects) are organized in a hierarchy of *bounding boxes* (actually bounding intervals) over the time dimension. This requires no duplication of border elements anymore. In addition, it incorporates coarse summary information near the top of the hierarchy in order to provide a quick preview [CALG07, CGL00].

2.3.7 The Paraver Trace Format

The Paraver trace format [CEP01a] is another ASCII format that collects all event information in a single file. It is well documented in [CEP01a] but the read/write library is not available as source code. Unlike all other trace formats, the Paraver format defines only three event record types:

- state change events for enter/leave events of functions or more general regions (8-tuple),
- atomic user events with a key and a value, e.g. for performance counter values (8-tuple), and
- communication events for point-to-point or collective communication (16-tuple).

The records are encoded as tuples of integer numbers. All components may carry arbitrary values as far as the format itself is concerned. The format will transport any contents as far as the producer and the consumer agree on the semantics. Still there are definitions of standard identifiers that are used by the Paraver and Dimemas tools, compare Section 2.2.2.

Like all other formats, Paraver requires the event records to be sorted by time stamps, but the event records need to be sorted by event type as the secondary sort criterion. Furthermore, the format uses two fixed hierarchies to describe logical and physical location of events separately. This combination allows to track the dynamic process placement during the execution.

The *process model* for logical location consists of:

- threads as smallest entities,
- tasks containing multiple threads,
- applications containing multiple tasks, and
- workloads composed of multiple applications.

The *ressource model* for physical locations contains:

- CPUs as smallest entities,
- nodes of multiple CPUs, and
- systems of multiple nodes.

Besides the single events file, the Paraver format supports two optional files with complementary information. The so called *configuration file* allows to specify some "semantics" for the "semantic free" events. This includes display options, for example whether to show absolute or derived representations of performance counter values or which colors to use for certain states. The so called *naming file* allows to replace identifiers with names, for example for functions or states or any members of the process model and the ressource model.

2.3.8 The DeWiz Trace Format

DeWiz provides its own trace data format called *NOPE* which shares many similarities with the other formats. It uses a binary encoding and distributes parallel traces over multiple files.

Apart from the NOPE trace format, DeWiz includes an interesting concept that is connected with trace formats as well as with memory data structures. It allows to serialize the memory data structure into a generic serial byte stream. Later, it can be restored to the original memory data structure. The serialized form is suitable for transfer over communication sockets or network connections. It is intended to exchange data between components of the DeWiz toolset [BKN04, GUP03]. Yet, it can be used as a convenient trace file format as well, compare Sections 2.2.5, 2.4.3 and 2.4.5.

2.3.9 The TAU Trace Format

Although primarily focused on profiles, TAU privides its own trace file format. It is used by TAU's measurement facilities to generate traces that can be converted to third party trace formats. Furthermore, it can be used to extract phase-based profiles or flat profiles afterwards as shown in the TAU collaboration diagram in Figure 2.23.

The TAU trace format supports the most widely used event types and provides a support library for writing and reading. TAU traces consist of a small definition file and the potentially large event file. Traces of parallel processes or threads can be written to multiple local traces at first. Later, they are merged to a single global trace by the `tau_merge` tool.

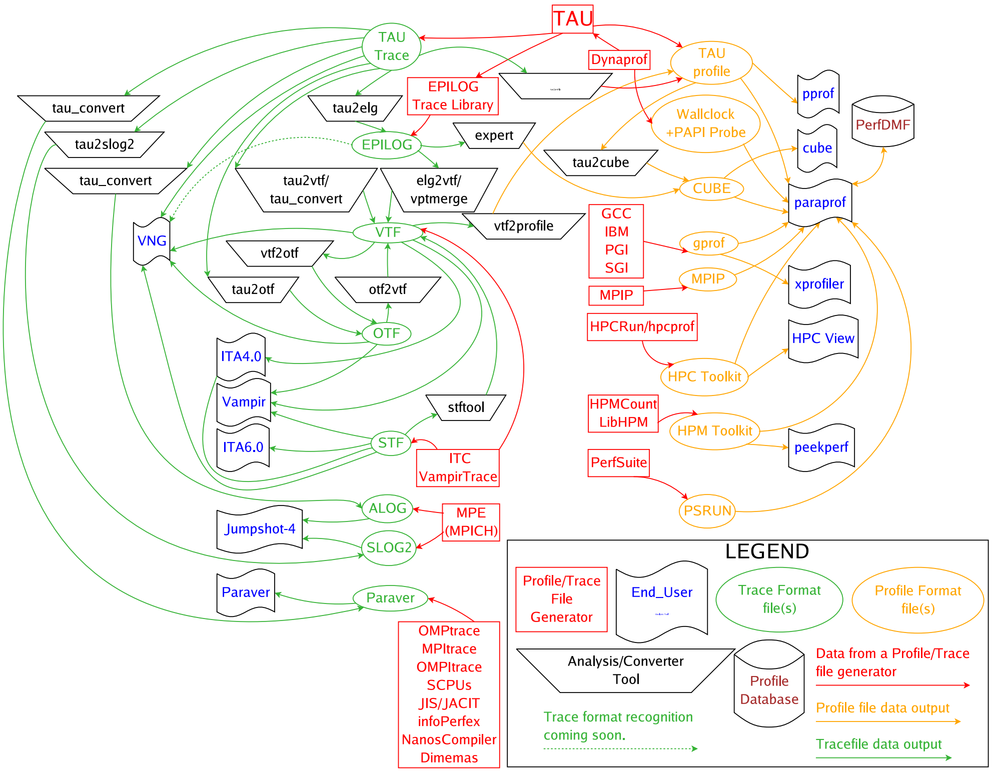


Figure 2.23: Collaboration diagram from the TAU documentation showing the interoperability of trace file formats and profiling formats as well as tracing and profiling tools. (Taken from [She])

The TAU format encodes the records in a binary and explicitly platform dependent manner, in particular the platform byte order is not compensated. All record types consume the same fixed number of bytes [She04]. This simplifies internal management but imposes a certain storage overhead and limits the extensibility.

2.4 Memory Data Structures

The memory data structures of the tools mentioned above are designed rather simple and straight forward. In particular, there are hardly any publications about the memory data structures of such tools. Instead they are regarded as a minor component of the overall software design. Fortunately, this information could be revealed through direct contact with the respective authors. For many tools access to the source code was granted [BMN08, koj05, She04, GUP03].

2.4.1 The Vampir and VampirServer Data Structures

The Vampir tool provides visualization of trace data as well as navigation. Furthermore, it offers statistic summaries according to the current time interval and process selection. All is presented to the user in an interactive manner. This means, trace data evaluation needs to be reasonably fast in order to give a truly interactive feedback upon user interaction.

Therefore, it is necessary to load the entire trace into main memory. Extensive external access to hard discs or network storage upon interactive user requests would be too slow. Even within main memory, trace evaluation becomes time-critical as far as very large traces are concerned. Processing a request for very large traces of 1 GB and above may require several seconds.

The major effort is caused by generating the display information from the trace event data. The graphical rendering is (almost) never critical, because the display data is bound to be small compared to the size of the trace data. This is necessary for two reasons: The screen or printer resolution for graphical rendering is limited and more or less constant. It will hardly exceed 10^4 pixels per dimension. And furthermore the human perception is even more limited and cannot discern more than several hundred elements. Therefore, the evaluation can be regarded as a filter that extracts a small but suitable graphical representation from a huge but incomprehensible set of event records.

The Vampir tool uses plain C arrays of compound data types as memory data structures for event records. For different record types, separate arrays with different basic types are allocated. Yet, the event records from parallel trace processes are kept in a single array. Within each array the events are sorted by time stamps like they are in trace file formats. This allows linear traversal and random access to be implemented very efficiently. But during creation repeated re-allocation is required because the eventual size cannot be predicted generally. Even though re-allocation is done blockwise in large segments it becomes a performance problem for very large arrays.

VampirServer uses the `vector` class of the C++ Standard Template Library (STL) which provides a flexible array-like data structure. Like in Vampir, separate data structures are used for different event types. The distributed storage of event records is achieved by mapping one or multiple trace processes to every worker process, compare Section 2.2.1. Within each worker there are separate data structures for different trace processes. This allows separate evaluation of parallel event records. Yet, evaluation of point-to-point communication events needs to consider associated send and receive events which would be placed at different locations. Therefore, send and receive events are assigned to the respective sender process instead of the original process.

Based on this storage scheme, the interactive visualization is subdivided into three steps: At first, a search operation for the array indices of the current time interval is performed. This is done via binary intersection search in the sorted arrays. In the second step, the array section in question is traversed linearly to collect the evaluation results. This may be either display elements or some kind of statistical summary information. In the final step, the actual drawing is done. It renders a graphical representation of the display elements or the statistics, compare Section 2.2.1. In case of VampirServer the display data is transfered from the server to the client application beforehand.

In this scheme all separate data structures can be evaluated individually. The results are composed only during the last step by appropriately arranging the display elements together.

2.4.2 The EARL Data Structures

The *Event Analysis and Recognition Library* (EARL) from the *Kojak* toolkit is not a trace analysis tool itself but a software component that is used by *Expert*. It allows convenient access to event trace data via the EARL API, independent from an underlying trace format. It is implemented in C++ and provides interfaces for C++ and Python [WM00b].

While common trace file format libraries restrict reading of events to the original sequential order, the EARL API provides a more flexible way of accessing event data sets. EARL allows sequential read-through with call back handlers as frequently found in trace format reading libraries and it provides references from certain events to particular related events. This allows to traverse the event data set in a more flexible way.

There are three kinds of references supported by EARL, see below. The references between events are provided automatically by EARL and may point either forward or backward in time or to parallel process traces [WM00b]. Furthermore, the call back reading can be combined with the access via references.

previos/next: Connects all events to the canonical predecessor/successor event according to the logical and temporal order of events in every trace process.

send/receive: Connects matching *send* and *receive* events from MPI point-to-point communication operations in different process traces.

enter/leave: Connects enter and leave events of function calls or from general regions that comply with the stack property (FIFO property), compare also Lemma 2 in Section 4.1.1. This reference always points backwards in time: For *leave* events it refers to the matching *enter* event. For *enter* events it points to the *enter* event of the surrounding function or region. By this means, the function call stack can be traversed upwards following the respective enter events.

This convenient scheme requires a certain scope of the event trace data to be kept in main memory data structures. During a sequential read-through of the trace events, EARL performes progressive file reading for the following events. As soon as events outside the current scope are de-referenced, EARL either re-reads previous sections or reads ahead for subsequent sections automatically.

The memory data structure for trace events consists of an list of a generic data type for event records. The different event types are modelled in a class hierarchy where specialized types are derived from general ones. EARL's event type class hierarchy is shown in Figure 2.24, see also [Wol04].

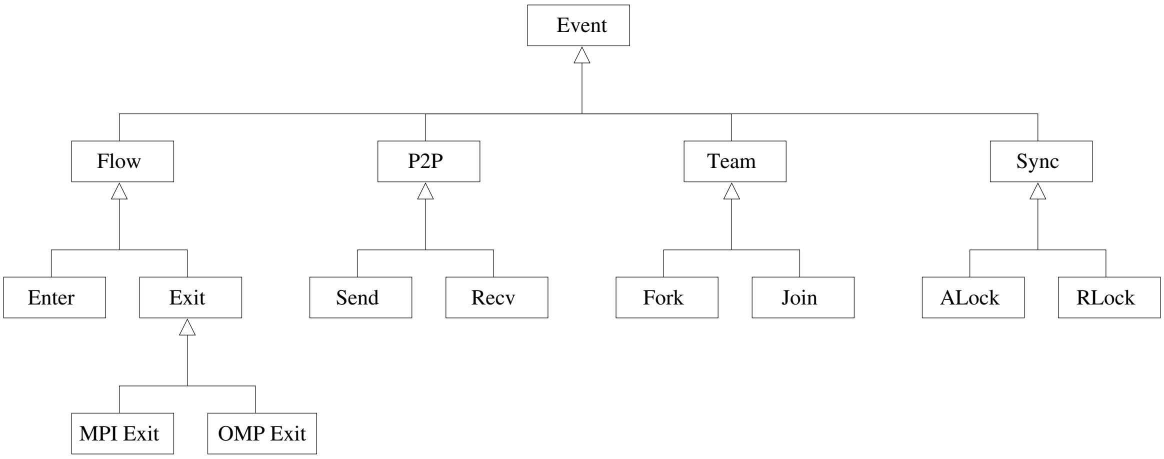


Figure 2.24: The event type hierarchy defined by the EARL library.

The EARL library provides a very convenient interface for navigation in parallel event trace streams. It is neither as restrictive as common trace format reading libraries that provide strictly sequential access only, nor does it allow random access, which would be more flexible but unnecessary for typical trace analysis algorithms. Instead, it serves as a suitable solution for mostly sequential read-through of events with occasional references to distant events. At the same time, it provides a good trade-off between overhead and flexibility [WM99, Wol04, WM00b].

2.4.3 The DeWiz Data Structures

DeWiz uses a very particular memory data structure that is closely related to the *Event Graph* definition in Section 2.2.5.

All events contain a global unique index $e = (p, s) \in I$ which consists of a process identifier p and a sequence number s within this process. Now, references between events are stored as pairs of indices $(e_i, e_j) \in I \times I$. Together, this resembles the mathematical definition of a graph $G = (V, E)$ with the set of vertices $V = R$ and a set of edges $E \subset I \times I$.

For representation of events (graph nodes) and references (graph edges) DeWiz uses an object oriented class hierarchy of data structures. Furthermore, it uses a common base class for events and references. Thus, references between events are modelled as a special kind of events that must not be subject to references themselves!

All events and references are stored in a common global container object, that provides a fast look-up operation for indices $e = (p, s)$. The container class as well as all event and reference classes allow serialization to a byte stream. This is usefull for the data transfer over network connections or for the persistent storage to files, compare Section 2.3.8. Furthermore, the data structure allows progressive serialization of accumulating event trace sets. This means, serialization and data transfer may start before the data is complete which is particularly favorable for online monitoring and analysis.

2.4.4 The TAU Data Structures

The TAU tools for analysis of event traces do not contain a real memory data structure like the previously presented tools. Instead of re-creating a spatial sequence of trace events in memory, it directly uses the temporal sequence as delivered by the common trace format libraries, compare Section 2.3.1. In particular, this requires processing of trace events strictly in the order of appearance.

This is a limitation for general trace analysis algorithms, yet it is sufficient for a number of algorithms. This scheme is used in the TAU trace tools but also in many other tools included in trace format libraries or run-time trace recording libraries. First of all, it is suitable for algorithms processing events separately and require no context information from related records. One example for this type of algorithms is the extraction of simple statistics about event records, like number of records per type per process. Here, the temporal order of events is irrelevant. Other examples can be found in the trace format conversion tools which perform a record-by-record translation from one format to another, e.g. `tau2vtf`, `tau2elg`, `tau2slog2`, and `tau2otf`. Here, the order of events is relevant but it is not altered.

For analysis algorithms that read trace events in the original temporal order, an extension to the previous scheme allows to incorporate some context information. This requires two conditions:

1. The amount of context information is reasonably small at any point in time. In particular it should be much smaller than the event data.
2. Context information needs to be collected early. This means the context information is extracted from some event records before it is consumed in conjunction with later event records. And it needs to be decided a priori which context information to collect from which records.

From the second condition follows that a *single pass* algorithm can only use *backward context*. An example is the summarization of run-time per function. It requires for every leave event the start time of the associated enter event as context information. This can be provided most efficiently with a replay of the function call stack with annotated timing information. It consists of a reasonable small set of backward context data and, in particular, it is not static but allows to add and remove pieces of information dynamically.

In order to support *forward context* as well, the previous scheme can be extended to a *two pass* algorithm. Now, context information can be collected in the first pass, irrespective whether it is before or after the recipient. Filter operations are typical examples: The filter rules are established in the first pass (e.g. the most frequently called functions) and applied in the second pass (e.g. ignoring events related to that particular functions).

A further extended *multi pass* scheme is not used by any of the investigated tools and seems undesirable, because the repeated I/O of potentially huge trace files will cause substantial overhead.

2.4.5 Similarity to Trace File Formats

When comparing trace file formats with memory data structures for event records it becomes evident that both are very similar. The two basic design concepts are shared between both:

- record types and
- sequential containers.

On one hand, there are *record types* as the smallest units of information, compare also Section 2.3.1. In trace files every record with its individual set of properties is encoded separately while in memory data structures, all record types are mapped to corresponding data structures or classes.

On the other hand, there are sequences of records in temporal order. In trace formats this corresponds to a single trace file or to multiple files, compare Sections 2.3.3 and 2.3.4. In memory data structures it can be found as a single list (e.g. EARL, compare Secton 2.4.2) or as several arrays for different record types (e.g. Vampir, see Section 2.4.1) or as multiple distributed lists for different record types of separate trace processes (e.g. VampirServer, see Section 2.4.1). DeWiz does not use lists or arrays as the container data structures, but provides the concept of a sequence of records by means of the sequence numbers of events, compare Section 2.4.3.

A further indication, that the common trace file formats and the memory data structures are closely related, is the fact that the back and forth transformation between both is rather simple and is usually done record by record.

2.5 Access Methods to Event Data Structures

Based in sequential data structures for trace events there are five types of typical query algorithms that all state-of-the-art tools rely on. At first, there are two basic algorithms for traversal and navigation:

- Sequential Iterator and
- Time Interval Search.

Then, the following two classes of algorithms provide (interactive) visualization for traces:

- Statistic Summaries over Event Properties and
- Timeline Visualization.

Finally, there are algorithms assisting the human user with tedious search for performance flaws:

- Automatic Analysis.

Like the memory data structures also the evaluation algorithms used by the presented tools are very similar, in particular the data structure access. This does not include actual drawing and rendering or user interaction schemes which are not interesting with respect to data structure design.

2.5.1 Sequential Iterator

The sequential iterator is the most basic access method. Beginning with any given position in a single stream of trace events it is capable of advancing to the following position i.e. the next event (*forward iterator*). A *reverse iterator* advances in the opposite direction, proceeding from a current position (event) to the previous one. Bidirectional iterators can advance either in forward direction or backwards. An iterator will indicate if it cannot proceed because the end of a stream is reached. For an implementation of a forward iterator on an array or linked list the position is represented by a position index or a pointer. A forward/backward iteration step is performed by updating the position based on local data. This is a very simple and efficient operation.

By means of continued iterator steps a stream of trace events can be traversed from an initial position until a termination condition becomes true or until a final position is reached. This can be used in order to implement routines for higher level evaluation:

1. access every event record exactly once
2. replay events in original order
3. replay function call stack
4. match associated *enter* and *leave* events
5. match related message *send* and *receive* events

The basic iterator is suitable to access every event record exactly once. This is useful when generating statistics about events, for example number of function calls, messages, etc.

Furthermore, it preserves the original (or opposite) order of events. This allows statements about causality of events without referring to time stamps which is particularly important for identical time stamps of successive events due to coarse timers.

2.5.2 Time Interval Search

A second support routine frequently used by higher level analysis algorithms is the search for a specified time interval. Through this means, a general analysis scheme can be applied to a well-defined sub-set of trace events. This is most useful to analyze certain phases of program runs separately.

Time interval search has to find positions p_a and p_b of earliest and latest event records with given time stamps $a \leq b$ in a single event stream. It identifies the position of the time interval $[a, b] \subseteq [min, max]$ where min and max are the minimum and maximum time stamp in the respective event stream.

In order to accomplish this, the search interval is truncated to $[a, b] \cap [min, max]$ first. If this results in an empty set the search operation will abort. Then, a modified binary intersection search is started which handles upper and lower interval bound simultaneously. Thus, the search needs $O(\log n)$ effort for n events provided that random access is possible.

Random access is possible for arrays and vector data structures but not for plain linked lists. For the latter case binary search can be enabled if an array of *anchors* is pre-generated. *Anchors* simply provide references/positions of an arbitrary subset of all list elements where the anchors need to be sorted with respect to their targets' ordering. The number or granularity of the anchors can be chosen adaptively. Then, the binary intersection search operates on the array of anchors, identifying the list elements closest before or after the actual interval bounds. From this position a *short* linear search is started, where *short* means not longer than the maximum distance between adjacent anchors' targets. For a list of n events and m evenly distributed anchor position, the search effort is reduced from $O(n)$ to $O(n/m + \log m)$.

For short distances a linear search operation might be a benefit for random access data structures, too. If the event of interest can be assumed to be close to the current position because respective time stamps have a small difference (compared to the maximum time span $max - min$) the actual execution speed could be increased by this.

2.5.3 Statistic Summaries

One of the most important classes of high-level evaluation methods is the computation of statistic summaries over various properties of trace events. In general, such methods produce a more or less constant result set from an arbitrarily large sequence of trace events. For example, if the number of occurrences of all functions (within time interval $[a, b]$) is computed, the result will be a list of one value per function (absolute or percentage). The length of this list is bounded by the number of functions in that trace. For alternative time intervals $[a', b']$ single functions might be added to or removed from this list. However, the list's length will not scale with the intervals length or the event count in a time interval in general.

There are three sub-classes of statistic summary algorithms, with different constraints:

1. evaluate on single events only
2. evaluate pairs of associated events in same stream
3. evaluate pairs of associated events in different streams

Single Event Statistics

The first and simplest class of statistics can be derived from single events, for example the number of messages sent or received on a given process. To compute such statistics, every event in the selected time interval has to be accessed exactly once and not necessarily in temporal order.

Local Statistics with Associated Events

Secondly, there is the class of statistics about *program states* which are defined by the pairs of associated *enter* and *leave* events in the same stream (process). Every state happening during the selected time interval needs to be investigated once only. The resulting statistic is computed as the sum over all occurrences of a state.

In order to achieve this, only *leave* events trigger the investigation of the state in question. The associated *enter* event (respectively its properties) needs to be available at this stage. By means of call stack replay this can be provided with minimum overhead.

Special cases for function calls that intersect with the selected time interval $[a, b]$ have to be handled separately. In order to correctly evaluate all possible intersections the complete call tree information at time a is required beforehand. It might be available as auxiliary information at certain anchor positions, compare time interval search in Section 2.5.2 and Open Trace Format in Section 2.3.4. Otherwise, only function calls with either *enter* or *leave* inside time interval $[a, b]$ can be covered.

Unlike for evaluation of single events, the order of event traversal is important to achieve an efficient algorithm. It allows stack replay and thus a fast reference from a current *leave* event to the associated *enter* event. Because the sequential iterator (see Section 2.5.1) provides this feature, the second class of statistic summaries can take advantage of the same basic algorithm as the first one.

Remote Statistics with Associated Events

The third class of summary statistics needs to match associated events from different streams. For example, for statistics about point-to-point message speed it is necessary to determine the message size as well as *send* and *receive* time. The latter two are not provided by a single event but by two independent events in different streams (respectively processes).

In order to match associated *send* and *receive* events, all parallel streams (process traces) need to be taken into account. Again, the replay algorithm is based on the sequential iterator (see Section 2.5.1). However, it is no longer possible to process the streams independently like for the previous two classes of statistics. For every process/stream there is a queue data structure (FIFO list) to keep a set of pending messages. If a *send* event is encountered it is added as a new pending message for the relevant peer process (receiver). If a *receive* event is found, it is compared against all pending messages for this process. The first positive match is identified as the valid peer event and removed from the list of pending messages. Matching of messages has to consider sender and receiver processes, MPI communicator and MPI message tag according to the MPI Standard [For95, For97].

The order in which to advance through parallel streams is only important for temporary memory requirements, i.e. the length of pending messages lists. A static or adaptive round robin scheme is suggested. When advancing parallel streams in an alternating way another source of ambiguities is introduced. Events with identical time stamps from different streams can be arranged arbitrarily in the analysis stream. This is by no means a restriction of parallel trace analysis but reflects a fundamental principle of concurrent programming [Lam78].

Again, there are special cases to consider when only one of two remotely associated events a and b is included in the current time interval T. The matching algorithm depends on the previous states of the pending messages queues, which need to be provided at time a. This might either be provided at certain anchor positions as pre-computed information (compare Section 2.5.2) or it needs to be generated by processing the trace from the beginning.

Even though the effort for this class of statistics will be linear with respect to event count, the latter case would be a severe performance disadvantage for small time intervals. Instead of the event count in $[a, b]$ the (potentially much larger) event count in $[0, b]$ would determine the total evaluation effort.

2.5.4 Timeline Visualization

Timeline visualization is the second most important high-level evaluation method, compare Figure 2.2. It consists of two stages: Computation of display data which is already adapted to the target pixel resolution and the actual drawing which is not important here, because it is independent from the trace data structures. The following is mainly dedicated towards the Vampir, see Section 2.2.1.

The computation of display data is closely dependent on trace data structures. Its result is used as input for the latter stage and can be stored in any convenient format. In particular, intermediate drawing data is rather small as it contains only a limited set of items that can be rendered with a given horizontal pixel resolution. The available pixel resolution is always considered constant and relatively small in the order of magnitude of 10^3 to 10^4 in the vertical and horizontal dimensions. It will hardly ever scale to the number of events in today's extensive traces of 10^6 to 10^{10} in the foreseeable future.

Timeline visualization considers only a selected time interval T like statistic summary evaluation. An interactive timeline display can implement zooming and scrolling operations by changing time intervals of interest during successive timeline queries.

The timeline diagrams show four major visualization items:

- function calls as colored boxes,
- point-to-point messages as arrows,
- collective communication as intervals connected by lines, and
- performance counter values as piecewise linear function chart over time.

For all according events a corresponding item is placed on the display area of w pixels width by h pixels height. Time stamps of events are mapped to the horizontal axis. The horizontal position $p \in [0, w-1] \subset \mathbb{N}$ is computed from the pixel width w and the time stamp $t \in [a, b]$ with the current time interval $[a, b]$ according to the following formula, compare also Figure 2.25.

$$p = \left\lfloor \frac{t-a}{b-a} \cdot w \right\rfloor . \tag{2.1}$$

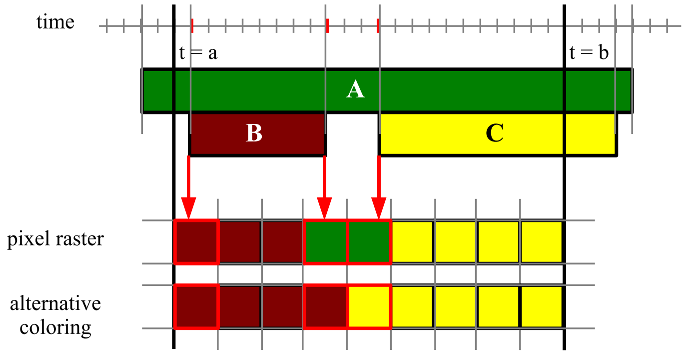


Figure 2.25: Mapping of time stamps to horizontal positions in the pixel raster (top). The transformation from the fine grained time stamps to the coarse grained pixels can be ambiguous (bottom).

Vertical positions are either derived from an event's process in global timeline diagrams or from call stack depth in local timeline diagrams. Both, processes and call stack levels are visualized as bars instead of thin lines. Usually, the vertical axis points downwards instead of upwards.

Function Calls

For every function call in the event stream a colored box is drawn at the appropriate vertical position which is computed from the time stamps of the *enter* and *leave* event according to Equation (2.1). The color is determined by the associated function, see Figure 2.25 for an example.

The drawing algorithm uses a stack which stores the drawing color associated to the function calls to be visualized. It keeps the current horizontal position p'. For every event it maps the time stamp t to the new horizontal pixel position p and draws a box from position p' to $p-1$ with the drawing color found on top of the stack. If the stack is empty, the background color is used instead. After this, the next drawing color is determined. If the current event is an *enter* event, then the drawing color associated to the respective function is pushed to the stack. If it is a *leave* event, then the current top element is removed from the stack, activating the previous drawing color again.

Special consideration is necessary for function calls that map to a horizontal width smaller than one pixel. Such function calls are ignored because they would be invisible anyhow. Thus the effort for calculating and storing that information can be saved. If there are several consecutive function calls smaller than one pixel that account more than one pixel's width together, then one of them is selected for display. This can be chosen according to the longest time share among those function calls or randomly, as illustrated in Figure 2.25. Based on the above drawing algorithm such a situation can easily be detected by the test $p = p'$, which means new pixel position is equal to the previous one.

Point-to-point and Collective Communication Events

For *send* and *receive* events an arrow is drawn from the *send* event's position to the *receive* event's position on top of the function call boxes, see Figure 3.9. Matching of associated events is assumed to be pre-computed by the same algorithm used with statistic summary queries (Section 2.5.3). Like for *enter* and *leave* events, horizontal positions are calculated from events' time stamps. Vertical positions are determined by process association or call stack level.

Collective communication events are visualized similar to *send*/*receive* events. Instead of a single arrow the events in neighboring process bars are connected with two lines. The first line connecting both start points, the second connecting the end points. For multiple process bars visualized next to each other this results in a vertical polygon covering the collective communication operation over all participating processes, see Figure 3.10. Again, special handling is necessary for spots with more communication events than horizontal pixels. Drawing of single arrows or lines is suppressed in this case. Instead, a marker is placed showing that there is a large quantity of communication events. It signalizes the user that individual events will be revealed after sufficient zooming to the respective time interval and thus, increasing the horizontal pixel resolution.

Performance Counter Samples

Performance counter values over time are visualized separately from the other events. The alignment to function call states or other events is achieved by separate mapping from the same time stamp resolution to the common pixel raster, see also Figure 2.25.

Performance counters are always queried in discrete resolution. The intermediate behavior between successive counter samples cannot be retrieved. Instead, it is either interpolated linearly for properties that are accumulated continuously or it is assumed to be constant for properties that are pinpointed at particular values. Furthermore, for the representation of differentiated performance counter values over time, the difference quotient from successive counter samples is used, which is interpolated with a constant function over time. Therefore, the performance counter display always shows piecewise linear or piecewise constant functions over time.

Computational Effort for Timeline Visualization

The computational effort for the generation of drawing data, excluding the actual drawing operation, is linear with respect to the number of events $O(n)$ in the current zoom interval $[a, b]$. This is true for all four visualization items included in the timeline display and, thus, for the sum.

For small traces or limited time intervals of large traces this is no challenging task. Yet, visualization of the total time interval for large traces becomes computationally expensive. Interactive response to user interactions cannot be guaranteed. At the same time, the drawing area has constant size $w \times h$. A visualization scheme, that scales with $O(w \cdot p)$ or $O(w \cdot h)$ and that is independent from the event count, would be most convenient. However, such an approach is not found in any of the existing tools and cannot be achieved with the traditional data structures.

The speed of the actual drawing operation depends on the particular display type. It is supposed to have linear computational effort with respect to the amount of display data. Therefore, it has also linear effort with respect to the display size, because both are proportional as stated in Section 2.2.1, in Section 2.4.1 and in Section 2.5.4 above.

2.5.5 Automatic Analysis

There are various approaches for automatic performance analysis that try to detect known performance flaws in order to relieve this task from the user. The general scheme consists of three steps:

1. find certain pre-defined *situations* in the event stream
2. survey each occurrence of a situation, rate it as critical or non-critical
3. report a set of most critical situations to the user

The first part is the detection of *situations of interest*. *Situations* are characterized by a group of related trace events but not by a single event. Usually, the search algorithm traverses the event trace in temporal order trying to match all potential situations to the current events. The actual structure of a situation needs to be explicitly defined, either encoded in the detection algorithm or as a formal specification [WM00c, FGM$^+$01]. An example situation is the exchange of a point-to-point message. It consists of the events `enter MPI_Send`, `send` and `leave MPI_Send` on the sending process and the associated events `enter MPI_Recv`, `recv` and `leave MPI_Recv` on the receiver process.

As a second step, each situation positively matching the specifications is surveyed for performance properties. According to this, it is rated as critical or non-critical or alternatively with a continuous *severity* value $s \in [0, 1]$. For the point-to-point message situation from the above example, the rating might consider multiple aspects. It might be classified as a *Late Sender* event if the `enter MPI_Send` event happens a considerable amount of time after the `enter MPI_Recv` event. If the `enter MPI_Send` event happens much earlier than `enter MPI_Recv` it is rated as *Late Receiver*. The severity value of either case would be determined by the actual amount of the delay.

Detection as well as rating of situations requires previous knowledge about semantics and about critical and non-critical behavior. Therefore, only well-known types of situations are considered for automatic analysis. Furthermore, only for frequently occurring situations the effort for specification and for automatic detection is worthwhile. Because of this, automatic performance analysis focuses on standard situations which have well-defined semantics and performance behavior. First of all, parallelization paradigms are targeted which includes MPI calls as well as OpenMP directives.

Application specific function calls are rarely or never subject to automatic analysis because both, semantics and expected/optimal run-time behavior are unknown. The required background knowledge is in general unavailable and cannot be provided in a universally valid way.

Finally, the results of automatic performance analysis will be presented to the user. For this purpose the critical situations found are grouped and sorted. At first, sets of similar performance flaws are reported only once instead of for each and every repeated instance. Then, they are sorted according the associated severity value s in order to show most critical situations first or in highlighted style [WM01, FGS03, GMT04]. In addition, the user is provided with source code locations causing critical situations.

All of the algorithms for automatic performance analysis can be transfered to the CCG data structure as well. The most straight forward adaptation would traverse the process traces in temporal order providing a replay of all events, compare 3.4.2.

2.6 Event Trace Compression by Statistical Clustering

Only few approaches for compression of event traces for parallel performance analysis have been documented, even though the SPMD (single program multiple data) paradigm suggests redundant behavior in parallel processes. One exception is the approach for Roth and Nickolayev et. al. [Rot95, Nic96, NRR97] who proposed clustering of process traces according to "‘similarity"’ of run-time behavior. For each data cluster only a single representative trace is kept. This scheme is well suited for SPMD-style parallel applications but cannot exploit redundancy due to repetition within a singe process trace.

For the classification into clusters, the single process traces are mapped to trajectories in a d-dimensional parameter space according to d performance properties $(p_i)_{i=1...d}$. The p_i may be comprised of either continuous or discrete properties. They are summarized over a sliding window of the (temporary) process traces. This equals a low pass operator filtering high-frequency effects which would otherwise inhibit this compression scheme. Then, the clustering is computed according to the Euclidian distance of the trajectories in certain phases. In order to respect changing behavior, a re-clustering is performed either in fixed intervals or adaptively.

The compression factor for a single cluster is always equal to the number of members. If there are multiple clusters for many parallel process traces then the total compression depends on the sizes of the representative traces of each cluster. For real-world examples with up to 128 parallel processes a total compression factor of 12 to 90 has been reported.

This approach has been extended for real-time compression [Nic96, NRR97]. While every process trace is recorded in a memory buffer, the clustering is computed by a central (external) instance. Then, only processes chosen as a representative will actually output the trace buffer to a trace file. All remaining processes will refer to their cluster representative instead.

2.7 Memory Access Traces and Compression

The earliest references in literature about event tracing in general an about trace compression in particular are about memory access traces. Such traces contain very simple data, usually only in form of consecutive memory addresses. Few references consider three types of access, which are instruction fetch, data read and data write. Furthermore, the memory access traces are always regarded as purely sequential and they contain no timing information for the events.

The main purposes of memory access traces are analysis, simulation and re-play of memory accesses for the optimization of caching methods. It is used in the design process of hardware caches, cache hierarchies, and their replacement strategies and it is most usefull for the analysis of virtual memory paging algorithms and replacement strategies, which are implemented in software.

The compression methods for memory access traces can be divided into two groups. The first group achieves data compression by selectively omitting data that is not needed for a particular purpose. Therefore, this are lossy compression schemes that do special purpose compression or *semantic compression*. The second group exploits regularity and repetitive behavior of the data in order to achieve lossless compression. Both, can also be combined to create an improved lossy compression scheme.

The former group of compression methods are associated to the simulation of a particular cache replacement strategy [KSW99, Smi77, Sam89, JH94, CR71, GC97, AH90]. For example [KSW99] is dedicated to simulation of virtual memory paging with LRU (least recently used) strategies with different parameters. It would require substantial changes to be suitable for other page replacement strategies. The compression transforms the original trace into an equivalent trace that reproduces the exact same LRU paging behavior [KSW99]. This allows substantial data compression, because multiple successive accesses to the same page can be removed. Under certain conditions this scheme is guaranteed to produce optimum compression, i.e. the shortest equivalent trace [KSW99]. Depending on the regularity of the examples the lossy compression methods allow data reduction of several orders of magnitude.

The latter group of lossless compression methods are related to general purpose lossless compression procedures, like Lempel-Ziv (LZ) or Lempel-Ziv-Welch (LZW) [ZL77, Wel84]. In a first step, the trace data is transformed into a so called *difference trace* [Sam89, JH94] (also known as delta coding). By storing the differences between successive addresses, usually the numeric values become smaller than the original addresses and can be encoded with less storage space. Furthermore, the transformed values reveal more regularity, for example the the widely used patterns of successive memory accesses with constant stride will be mapped to a constant sequence. This is very convenient for the second step, which uses standard lossless compression methods to effectively exploit the above mentioned regularity.

The lossless memory access trace compression has been reported to achieve total data compression by factors of 8 to 30. Further improvement can be achieved by handling instruction fetch accesses and data read/write accesses separately. With this extension compression factors of up to 150 have been observed [Sam89].

2.8 Compression of MPI Replay Traces

Müller et. al. presented an approach for compression of MPI replay traces in 2007 and 2008 [NMSdS07, RMdSS08]. Such traces contain only the information about MPI operations that is needed to re-create the original communication behavior. They cover no additional event types like function call events, user defined events or performance counter samples and carry no individual timing information for events.

The compression scheme consist of two stages, in order to compress process traces of parallel programs during run-time. The *intra-node compression* stage looks for repeated patterns in a sliding window over single event streams. Repetitions are mapped to so called *regular section descriptors* (RSD) consisting of a sub-sequence and a repetition count. The RSDs may be nested during this process. The following *inter-node compression* applies a post-processing to the single pre-compressed process traces. It replaces common RSDs in all or some parallel process traces with so called power-RSDs (PRSD) to achieve further compression. The latter step uses a radix-tree scheme to allow an efficient handling of very large process counts (MRnet [RAM03]).

The approach is reported to achieve compression in the order of magnitude of 2 to $> 10^3$. For favorable examples the authors even claim "near-constant" trace sizes when scaling with respect to the number of parallel processes [NMSdS07, RMdSS08], i.e. arbitrarily growing compression ratios.

A recent extension added timing profiles for the duration of computation phases between MPI calls which are not covered otherwise by this kind of traces. This will allow to replay the communication behavior with realistic durations between successive MPI calls. For every PRSD the time information is stored in a histogram with a fixed number of bins which allows to capture a coarse distribution of the timing behavior [RMdSS08]. It can neither provide individual duration information for single phases between MPI calls nor time stamp information for communication events.

3 The Design of the CCG Data Structure

This chapter introduces the contribution of this dissertation, which is a sophisticated memory data structure for event trace data named Complete Call Graphs. It presents the design of the data structure and the compression feature as well as adapted evaluation algoritms.

The new data representation is fundamentally different from the classic linear representations. And in contrast to the traditional manner, it allows significant data compression inside main memory which is completely transparent to read access which means, no explicit de-compression is required. Furthermore, this novel data structure allows the design of adapted query algorithms which provide an extra performance benefit in addition to the advantages of the smaller memory footprint.

3.1 Trace Data and Trace Information

Performance Tracing has a reputation for producing large amounts of data where *large* has always been defined by the time's standards. What are the reasons for the very big and ever growing trace data sizes? And how does the actual *information* contained in the data relate to this trend?

There are three main factors contributing to the growing amounts of trace data. The first is the trend towards more complex software projects with larger source codes, using many third party components and libraries. This also includes a growing code due to optimization and source code specialization [MWD00]. The second factor is the evolving instrumentation and measurement techniques recording more detailed data. More fine grained instrumentation produces more trace events and additional properties for the various event types increase the data volume even more. Typical examples are hardware performance counters or the cumulative load of the I/O subsystem. The third factor is the trend for longer and faster program runs as well as massively parallel execution. Both contribute to the enormous growth of trace data by repeating parts of a program more often. Usually, the number of repetitions of essential parts of a program directly relates to trace data size. This means the trace data volume is proportional to the iteration count as well as the degree of parallelism.

The last reason is certainly the main cause for the growth of trace data volumes. It also explains why the trace sizes keep up with the development of computing technology and available storage. Considering the three reasons for growing data volume, how does the actual *information* for the user increase? For larger software systems (first reason), additional data relate directly to valuable information about additional components. More elaborate instrumentation and measurement (second reason) also allow a better insight into a program's behavior. However, more repeated occurrences of some parts of a program (third reason) do not provide more useful information to the user, in particular, if a substantial number of repetitions has already been observed.

The example in Figure 3.1 illustrates this: There is a simple loop of iteration count n. Within the loop four function calls are performed in a fixed order. The four functions may contain further sub-function calls. A trace of this loop will give an account on two aspects. At first, the call structure, i.e. the order and hierarchy of function calls. Furthermore, the temporal behavior, i.e. the run-time consumed by every function call. If the code behaves regular, then a single iteration might be sufficient to analyze this part

```
for ( i= 0; i < n; i++ ) {

    foo( i, ... ) {
    bar( i, ... );
    communication();
    fubar( );
}
```

Figure 3.1: Example of a simple call sequence of four functions that are repeated n times.

of the code. Instead of looking at every iteration independently, a comparison of a certain number of iterations will reveal the average behavior. Apart from the average case, some outliers might occur either in the call structure or in terms of run-time.

In order to learn about the average case, a certain number of iterations will be sufficient as an adequate statistic sample. Larger numbers will not improve the confidence of statistics notably. Hence, there is no point in storing data about more than a few hundred or few thousand iterations even if there are millions of repetitions in one trace. Most of the repetitions will be nearly identical to the average case. Storing many identical iterations implies a high level of redundancy. Therefore, the amount of actual information is significantly smaller than the data volume. Yet, data about outliers, i.e. iterations that diverge from the average case, do provide additional information. By definition of *outlier* they appear infrequently and do not account for the majority of data.

It follows, that redundant data does not provide additional information and the actual information does not require large quantities of data. Thus, a compression scheme for redundant data could diminish the data volume of traces while preserving all of the essential information enclosed.

In the next section, the new CCG data structure for trace data will be introduced which allows efficient identification of redundant event sequences in a trace. The following section explains how to remove redundant parts in order to compress the data volume in a way that does not require explicit de-compression in order to read the data. The subsequent sections deal with persistent storage of compressed traces and adapted evaluation procedures.

3.2 Tree Data Structures for Event Traces

The basic scheme of the newly introduced event trace data structure is derived from general *Call Graphs* or *Call Trees* [GDDC97, GC00]. The most widely known form are the so called first-order call graphs, that summarize the caller-callee relation between functions. This is frequently found in basic program profiles [GKM82]. Higher-order call graphs do not consider a single caller function as the argument, but instead a hierarchy of n nested function calls as *call site* (n-th order call graphs) [MSM05]. Statistical properties can be appended to the nodes of a call tree in order to describe the average behavior of all function calls that match this node. This is used for example to produce more elaborate profiling results than plain *flat profiles* [GC00].

As a major difference, the new data structure does re-produce the *complete* function call hierarchy, not summarizing any information. Every function call is represented as a node, sub-function calls appear as ordered child nodes. This results in a tree graph, i.e. a directed acyclic graph, for a sequential program or for every single process of a parallel program. Hence, the new data structure is referred to as *Complete Call Graph* (CCG).

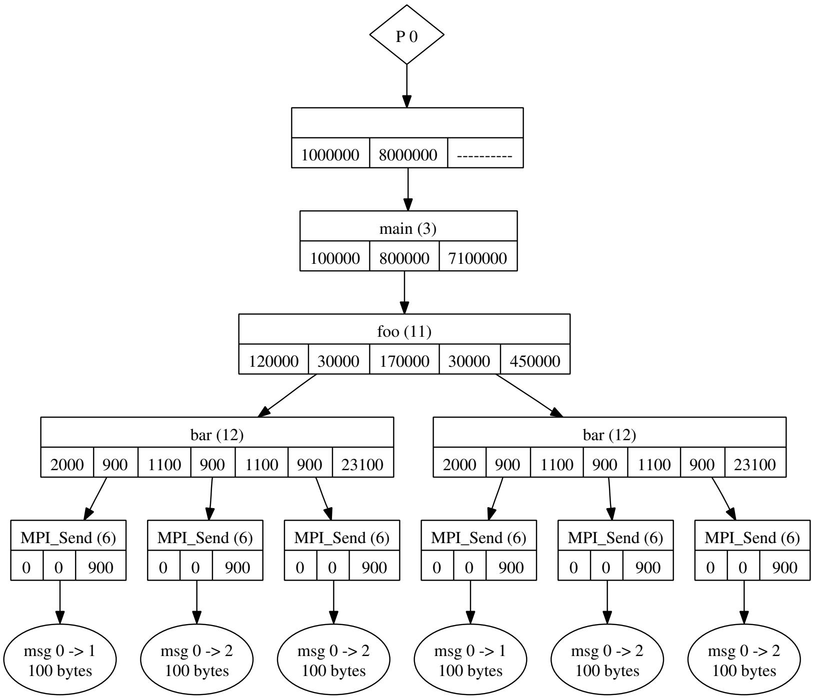


Figure 3.2: Example of a Complete Call Graph (CCG) of a single process. It shows a hierarchy of function calls to `main`, `foo`, `bar` and `MPI_Send` as well as atomic message *send* events at the leaf nodes.

Properties to function call events are annotated to the graph nodes, covering function identifiers, run-time information, performance counter values, and some more. Furthermore, this data structure is not limited to function call events but is capable of storing any kind of event along with the according properties. There are special node types for *atomic events*, i.e. all non-call events, like message passing events or input/output events. Atomic nodes are always leaf nodes, i.e. they have no children.

Thus, the overall graph structure is defined by the function call hierarchy of a program trace. All other information is appended to that graph. An example of such a graph is given in Figure 3.2. It shows the Complete Call Graph of a single process `P0` where the `main` function performs a single call to function `foo`. Function `foo` calls `bar` twice, which calls function `MPI_Send` three times. Every function call is represented by a node with corresponding name and run-time information (see also the following Section 3.2.1). Sub-function calls are denoted by child nodes. Topmost, there is an unnamed artificial function call node that ensures that there is always a single root in the tree. The leaf nodes are an example of atomic events shown as ellipses. They are child nodes to the `MPI_Send` calls and depict the actual *send* event of a point-to-point message which is separated from the according function call. The graphical representation reflects the memory data structure, where nodes correspond to instances of a C++ class which are connected by pointers.

The most important distinction of the CCG data structure from a classic flat data structure is the existence of a canonical and well defined hierarchy, i.e. a set of sub-structures (sub-trees) along with an inclusion relation. Based on this fact, redundant sub-trees will be identified and replaced with a reference to a single common instance. In particular, the hierarchy allows to do both very efficiently, see Section 3.3 for a more detailed explanation. But before the compression of Complete Call Graphs is presented, two transformations to the uncompressed CCG are necessary. This concerns the representation of the run-time information and the branching factor of the graph.

3.2.1 Time Stamps versus Time Durations

Alongside the general layout of the data structure the method of storing run-time information is crucial to all subsequent results. The classic way is to store time stamps, i.e. points in time relative to a fixed zero point, which is the start time of the program run. Usually, it is represented by an integer number that counts time in so called *ticks*, which are the smallest units of the available timer.

With regard to Complete Call Graphs using *time stamps* is most inappropriate, because by definition there can be no repetitions within a monotone increasing series of time stamps. The CCG uses *time durations* to express run-time information instead. Durations are specified as non-negative integer numbers in the same units of *ticks*. As shown in Figure 3.2, every node contains a list of time durations describing the run-time of itself and all child nodes. For every child node, the total duration is given adjacently to the durations of the gaps before/after each child call. This results in $2n + 1$ duration values for a node with n child nodes. Especially for leaf nodes, there is a single duration value that gives the duration of the according function call. This scheme introduces a new form of minor redundancy because the sum of all time durations belonging to a node is also stored within the parent node!

The transformation from time stamps into duration values is a sole subtraction operation. The reverse transformation involves additions of duration values while traversing a CCG in a top-down manner:

Lemma 1. *In order to restore the start timestamp t of any node N at depth level $k \geq 0$ of a CCG hierarchy one has to traverse the data structure from the root node to N. Assumed the path is $(n_i)_{0 \leq i < k}$, such that at level i the child n_i has to be entered with $0 \leq n_i \leq s_i$ and d^i_j is the $j'th$ duration value in the $i'th$ node on that path. Then*

$$t = \sum_{i=0}^{k-1} \sum_{j=0}^{2 \cdot n_i} d^i_j \tag{3.1}$$

will restore the time stamp t. Provided $n_i \leq s_i \leq b = \text{const}$ (see Section 3.2.2), the addition does not involve more than $2 \cdot b \cdot k$ terms independent of the total number of preceding events.

Proof. Induction on prefix paths. Assume the start time of the parent node t' is known. Then it follows $t = t' + \Delta t$, where

$$\Delta t := \sum_{j=0}^{2 \cdot n_{i-1}} d^{i-1}_j \tag{3.2}$$

is the sum of all durations in the parent node that precede the current node. Equation 3.1 follows from induction with the root node's start time $t = 0$.

□

Besides time stamps, traces may contain hardware performance counters with very similar characteristics that are also stored as monotone increasing integer values without explicit repetitions, for example floating point operations or cache misses. Therefore, performance counter samples that are aligned with time stamps should be transformed from absolute values into differences of successive samples, thus, appropriate performance counter samples will be handled like an *alternative time* specification.

3.2.2 The Bounded Branching Factor

The branching factor b is an important property of all tree-like graphs. It is the maximum number of children to any graph node. Obviously, it always needs to be $b \geq 2$ for non-trivial trees. Usually, the branching factor is not arbitrary but bounded by a constant. Above all, this avoids large trees degenerating into linear data structures.

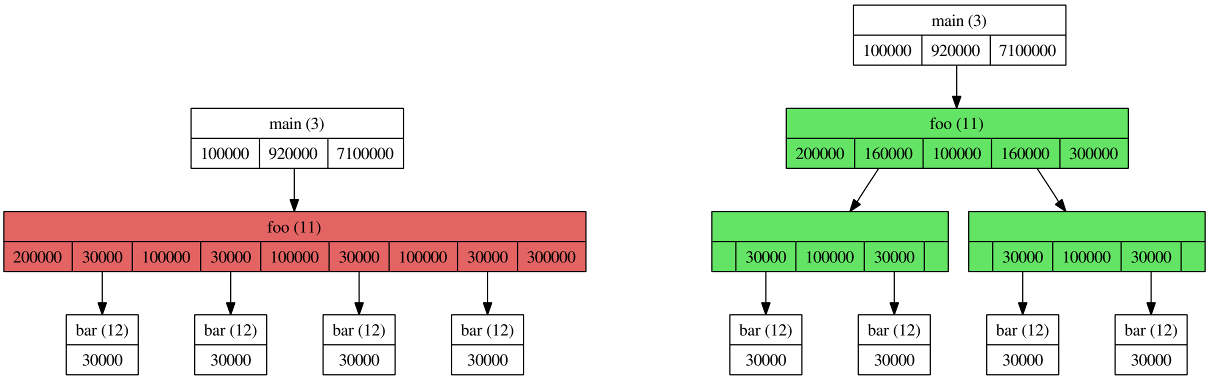


Figure 3.3: A wide graph node with more than $b = 2$ children (red) on the left is split up into an intermediate tree of artificial nodes (green) on the right.

The branching factor plays an important role in calculating the computational effort for certain operations on trees. For example, for simple search trees the minimal branching factor $b = 2$ is always optimal because it minimizes the effort for manipulation. This is one reason why computer science related literature mostly focuses on binary trees. However, binary trees are sometimes defined differently from a general tree that happens to have branching factor $b = 2$ [Knu97, Chapter 2.3]. More advanced search tree data structures favor larger branching factors. For example, B-trees and its variants use quite large branching factors. Interestingly enough, B-trees do not only limit the maximum number of children per node b but also the minimum with $b/2$ for non-root nodes [Knu98, Chapter 6.2].

In contrast to this, CCGs derived from actual program traces usually show very large branching factors which are unbounded in general. This is due to the fact that the maximum call depth of common (non-recursive) traces is rather small. Therefore, the resulting call tree looks flat. Furthermore, the call depth usually is almost independent of the traces size and more or less constant. Thus, if a trace is growing in size then the branching factor will increase, i.e. the CCG will become wider but not deeper.

The definition of the CCG data structure can be modified slightly in order to avoid this unfavorable behavior by splitting graph nodes with more than $b \geq 2$ children. From a single wide node with $n \gg b$ children, a set of artificial nodes is generated. Those artificial nodes are arranged in an intermediate tree with minimal depth. Figure 3.3 shows an example of a wide node and the result of the split operation.

The split operation is explained comprehensively in Section 4.1.2. The consequences for all query algorithms are limited to book-keeping issues, i.e. they must be aware that there are special nodes which do not represent an actual function call. This modification results in significant positive effects to the data compression ability and the speed of query algorithms, see the following Sections 3.3 and 3.4.

3.3 In-Memory Compression

As stated above, in-memory data compression is achieved by identifying and removing redundancy from a CCG. The general assumption is that multiple executions of a part of a program will likely produce equal or similar sequences in the trace. These will then be transformed into equal or similar sub-trees of a CCG. From the point of view of the CCG, sub-trees are regarded as redundant if there are multiple *equal* or *similar* instances. If there is a set of *equal* or *similar* sub-trees, all but one can be deleted and replaced by a reference to the remaining instance. Of course, maximum compression is achieved by replacing the largest possible sub-trees.

After this transformation the CCG is no longer a tree but only a tree-like directed acyclic graph. While most of the tree properties sustain, it is no longer true that every node has at most one parent node.

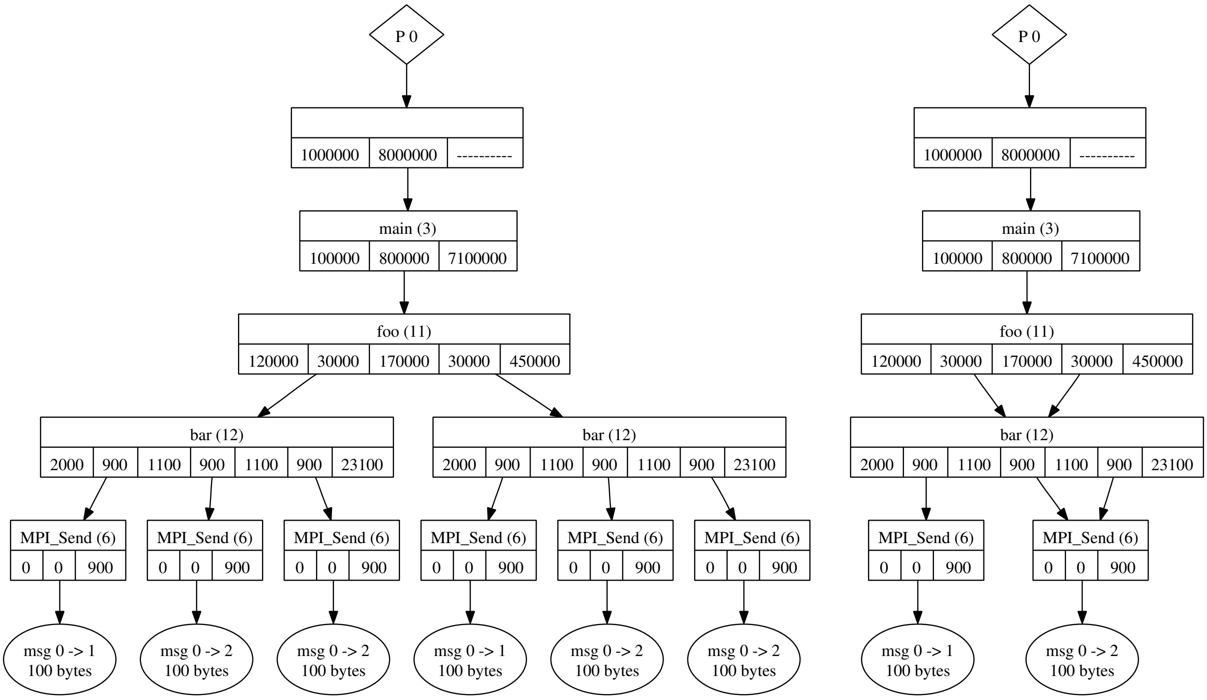


Figure 3.4: Comparison of an uncompressed (left) and a compressed Complete Call Graph (right). Both represent exactly the same information with 17 respectively 8 graph nodes.

Therefore it is called *compressed Complete Call Graph*. This is the reason why the data structure has not been named *Complete Call Tree* in the first place. As an example Figure 3.4 compares the plain CCG with the *compressed Complete Call Graph* of the same trace as in Figure 3.2. Originally, there were two redundant sub-trees: The calls to `bar` including all sub-calls are identical and thus redundant as well as the latter two calls to `MPI_Send` within function `bar`. However, they are not identical to the first call of `MPI_Send` because the associated leaf node specifies a different peer process. Both sets of redundant sub-trees have been replaced by references to a single instance in the compressed graph. This shows how compression of smaller sub-trees is nested into compression of larger surrounding sub-trees. As a result, data compression expresses itself in the reduced node count of 8 in the compressed graph compared with 17 in the uncompressed counterpart. Both representations carry exactly the same information.

For an actual implementation it is possible not to rely on the *unique parent node* property which disappears when transforming a CCG into compressed form. Later, a compressed CCG can be handled just like an uncompressed CCG, i.e. compression is completely transparent for any read access. Details can be found in Chapter 4.

The term Complete Call Graph (CCG) will denote a compressed graph in the following. This is the more general form, while the uncompressed graph is a special case therefrom.

3.3.1 Deviation Bounds for Soft Properties

As indicated above, the compression scheme can exploit the equality of sub-trees but also *similarity*. Thus, the approach may be implemented as a lossless compression scheme or as a lossy one. However, for the latter version, the errors or deviations arising are bounded and adjustable. From now on, the term *compression* is meant to be the lossy version if not stated otherwise. The lossless version is regarded to be a special case of the lossy approach with zero error bounds.

Unlike for *equality* there is no evident definition for *similarity* of CCG sub-trees. Therefore, a suitable definition is needed which takes the special requirements of event trace analysis into account. The desirable conditions for lossy compression include some properties that should not be altered at all, like for example identifiers, tokens and pointers. Those are called *hard properties*. Also, there are properties that allow small changes without much impact to their meaning. This implies that there is a proper metric to measure the difference between original and altered values. Timestamps, performance counter values or maybe even message lengths are examples for such properties, which are called *soft properties*.

Furthermore, for soft properties there must be reasonable bounds of deviations introduced due to lossy compression. In general, there can be two types, *absolute* and *relative* bounds. For any value v and its altered counterpart v' the absolute condition would be

$$|v' - v| \leq A \quad \text{or} \quad A_{lower} \leq v' - v \leq A_{upper} \tag{3.3}$$

and the relative condition would be

$$|\frac{v' - v}{v}| \leq R \quad \text{or} \quad R_{lower} \leq \frac{v' - v}{v} \leq R_{upper}. \tag{3.4}$$

where A and R respectively A_{lower}, A_{upper}, R_{lower} and R_{upper} are the given bounds.

The first and most important soft property of trace events is the run-time information. Here, both types of deviation bounds are relevant, always. At first, for every time stamp t_i it should be assured, that it is not moved to the future or to the past more than $T = const.$ ticks, i.e.

$$|t'_i - t_i| \leq T. \tag{3.5}$$

This is the absolute time deviation bound. Moreover, every duration of time between two time stamps $d_{i,j} = t_i - t_j \geq 0$ with $i > j$ must not be stretched or shrunk more than a given percentage:

$$|\frac{d' - d}{d}| \leq D. \tag{3.6}$$

This is the relative time deviation bound. Regardless of the representation as time stamps or time durations, both deviation bounds must be taken into account for temporal accuracy.

Now, two CCG sub-trees are defined as *compatible* if all hard properties are equal and the deviations in soft properties are below the respective deviation bounds. It follows that for two compatible sub-trees all respective child sub-trees have to be compatible, too.

3.3.2 Sub-Tree Comparison

The principle of data compression is replacing compatible sub-trees. Yet, comparing sub-trees for compatibility naively imposes the cost of comparing all nodes in it, that means completely traversing all sub-trees. Thus, the computational effort would be proportional to the number of nodes in a sub-tree. At the same time, nodes will be touched multiple times when comparing nested sub-trees. In order to minimize this effort the suggested approach will introduce a sophisticated sub-tree comparison scheme. It considers the root nodes of sub-trees only and does not take child nodes into consideration directly.

Rather, it restricts the order of processing sub-trees. A sub-tree's root node must be processed after all of its child nodes. Instead of checking child nodes recursively, the child pointers are compared: If two corresponding sub-trees have been found to be compatible, they would have been replaced by references to the same instance before. Thus, the child node pointers in both parent nodes would be equal. However, if two corresponding child node pointers are different, then the according sub-sub-trees are assumed to be incompatible. Thus, the enclosing sub-trees cannot be compatible either.

The pairwise comparison of all local properties of two graph nodes requires constant effort $O(1)$. The check for pairwise equal child node pointers causes linear effort with the number of direct children. If the branching factor is bounded by a constant $b = constant$ (see Section 3.2.2), then the computational effort of $O(b)$ for the node-by-node comparison can be regarded constant as well.

Time Deviation Estimation

After the sub-tree comparison has been simplified to a node-wise scheme, the deviation bounds for soft properties still need to be guaranteed globally. For some soft properties this can be achieved on a per-node basis, for others extra effort is necessary. For the run-time information, which is the most important example of soft properties, there are two deviation bounds (3.5) and (3.6) which need to apply globally. Therefore, the following method is explained with the example of run-time information but it is applicable for other soft properties as well.

In order to ensure relative deviation according to (3.6) it is sufficient to locally guarantee the bounds for every atomic duration value d stored in every graph node. Then it follows from additivity that the condition holds globally, i.e. it is true for every duration value between arbitrary time stamps, that can be expressed as a sum of atomic duration values.

For the absolute deviation bound (3.5) there is no local criterion that guarantees the global condition. Because run-time information is stored in terms of duration values, the original time stamp t and its deviated counterpart t' need to be recomputed from local terms of various graph nodes – compare Equation (3.1) and Figure 3.5. Thus, total deviation accumulates from various local deviations. When deciding whether a local deviation is acceptable or unacceptable, previously introduced deviations in current node's subtrees need to be considered (bottom-up approach).

See Section 4.2.4 for an efficient algoritm that propagates accumulated deviation of sub-trees from child nodes to parent nodes. By using interval arithmetic for the deviation values it allows the cancellation of positive and negative contributions.

Influence of the Branching Factor

Besides its effect to the computational effort of CCG construction (compare Section 3.2.2), the limited branching factor is important for the overall compression achieved. In general, the ability for compression increases with smaller b. A higher child count causes graph nodes to carry more single properties, i.e. more values to compare. All respective properties must be compatible for two nodes to be compatible. This includes all properties concerning direct child nodes. If $p \in [0, 1]$ is the probability that any two corresponding properties of two nodes match and all properties are statistically independent, then

$$P(b) = p^b \cdot P_0 \tag{3.7}$$

estimates the probability for two complete nodes to be compatible. Since $P(b)$ is monotonic decreasing the compression ability will increase with smaller values for b. Correspondingly, more direct children per node make it more likely that at least one is incompatible with its counterpart.

3.3.3 Computational Effort

The computational effort for the outlined compression scheme for Complete Call Graphs comprises of several parts. For replacing a sub-tree X, all available sub-trees need to be searched for a compatible replacement. The available set consists of all previously encountered sub-trees that were not removed in favor of an existing replacement itself.

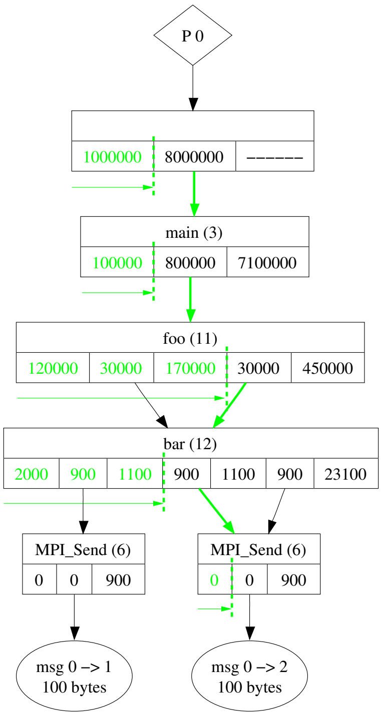


Figure 3.5: Accumulation of absolute time stamp deviation over multiple nodes. In order to re-create the time stamp of the 5th *send* event (marked green at bottom right) time differences on various (parent) nodes need to be considered (shown in green). All terms might introduce additional deviation affecting the particular time stamp and need to be bounded by adapted local bounds.

At first, a sub-set of those candidates is selected that match in respect of all hard properties. This is possible with $O(1)$ effort by means of an appropriate hash data structure. Then, the earliest of the remaining candidates is selected that matches with respect to all soft properties. Therefore, the soft properties of X need to be compared with that of every candidate. A single comparison operation of two sub-trees requires $O(b)$ effort, compare Section 3.3.2. If l is assumed the maximum number of replacement candidates for every sub-tree, the total effort connected with a single sub-tree replacement will be $O(l \cdot b)$. It is possible to have l bounded by a constant, see Section 4.2.2.

In order to compress a Complete Call Graph, the replacement operation needs to be performed for every single sub-tree. The number of sub-trees equals the number of nodes N in the uncompressed Complete Call Graph as every node is root to its associated sub-tree. Therefore, the total effort for CCG compression results in

$$O(N \cdot l \cdot b) \tag{3.8}$$

with b constant and small. The compression algorithm is discussed comprehensively in Section 4.1.

3.4 Customized Analysis Algorithms

In the beginning the CCG data structure is yet another way of representing data from program traces. The ability of in-memory compression provides a significant advantage over the classic storage scheme. However, when it comes to evaluating this data, this approach still has to prove its suitability.

3.4.1 The Conservative Approach

While the CCG approach is fundamentally different from the classic storage scheme it is possible to apply the classic evaluation algorithms by emulating a linear data structure from a CCG. This is a generic conservative approach, which is only suggested for an easy transition.

Only two basic operations are necessary for this purpose. They are both applicable for the classic data structure: a linear iterator and a fast search operation according to timestamps. The classic iteration operator traverses a list or an array stepwise forward or backward. It has a linear complexity of $O(k)$ for k steps. The classic search operation can be implemented as a binary intersection search (which is possible in arrays) or at least as a linear search (which might be inevitable on linked lists). At the best, the search operation features a logarithmic complexity of $O(\log N)$ for N events.

With the iteration and search operations provided, all classic queries could be adapted to the CCG data structure, see next Section 3.4.2. When applied to compressed data structures, they operate on a smaller working set of memory. Even though the query algorithms keep their computational complexity, this might yield a notable performance advantage. It might be a major benefit, particularly when external memory is concerned. This is especially the case, if the compressed data structure can fit into main memory as a whole while the uncompressed data cannot. However, improved algorithms can be designed for most significant queries to take further advantage of the CCG data structure.

3.4.2 Event Access, Iteration and Searching

This section covers the fundamental evaluation operations on traces which are:

- accessing single events and their properties,
- linear iteration in event sequences (process traces) in temporal order,
- time stamp search in (sorted) event sequences, and
- search according to arbitrary event properties.

The access to single events is a basic and simple prerequisite for any analysis and requires no principle changes. Either with classic sequential container data structures or with the proposed tree-like data structure, this is simply an access to elements of compound data types.

A linear iterator for the CCG data structure has to traverse the events in temporal order, which is different from traversing the nodes. For every event of the current graph node the succeeding or preceding event is contained in either the same node, the parent node or one of the child nodes. Migrating to a parent, child or neighbor node is a local operation, which can be assumed to have constant run-time. Thus, the complexity for iterating linearly through a CCG is $O(N)$ events stored in n graph nodes with $N \gg n$.

An efficient search operation for a timestamp t in CCGs can be achieved in a straight forward way. A recursive search is performed starting with the root node of a process graph. It is assumed that t is included in the time interval of the root node. Every graph node covers a time interval that is a superset of all child time intervals. Either, there is a unique child time interval that includes t, then is is made the new current node. Or no such child node is found, then the current node itself covers timestamp t and the recursion is terminated. In case the time stamp t does not match an actual event, then the nearest preceding or succeeding event can be returned optionally. With the branching factor b and maximum tree depth d the computational effort is $O(b \cdot d)$. Assumed b is constant and $d = O(\log N)$ the total complexity is $O(\log N)$, compare also Section 3.2.2.

Searching for events that fulfill arbitrary conditions can be carried out by a complete search. This can be done either by iterating through process traces linearly or by traversing the management data structure that contains all replacement candidates, compare Sections 4.2.1, 4.2.2, and 4.2.3. Both have linear computational complexity, either $O(N)$ with respect to the uncompressed node count or $O(n)$ with respect to the compressed node count $n \ll N$.

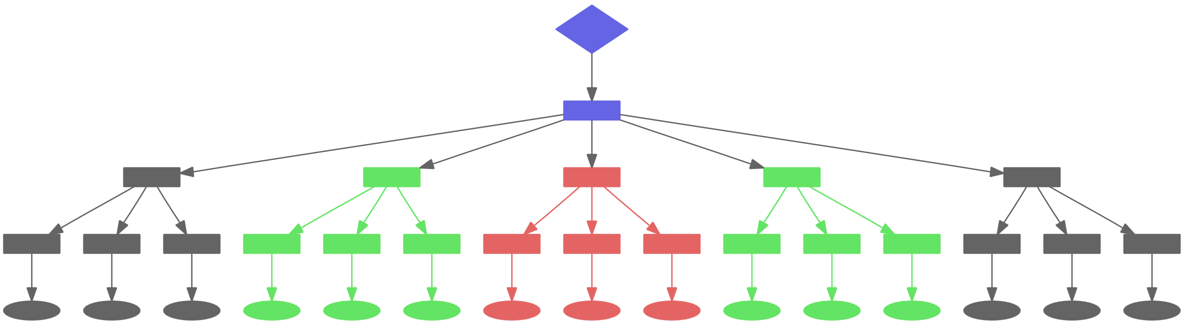


Figure 3.6: Schematic diagram of successive queries onto a CCG. An initial query involves all colored nodes. The intermediate results are cached. A successive query concerns only the red nodes. Now the intermediate result for the top level red node can be taken from the cache, all remaining red nodes need not be accessed again.

3.4.3 The Cached Summary Query

The most important variety of queries onto trace data is the collection of so called *summary statistics* for arbitrary intervals of time and process sets. Frequently encountered examples are the exclusive run time per function, messages per process or the time share of communication operations. In principle, such summary statistics can also be generated by profiling, even though profiling is not capable of producing this information for arbitrary time intervals.

The classic way to compute summary statistics is to iterate through the list of events and sum up the result over all single events. An alternative way of evaluation is possible by using the *additivity property* which states that the sum over values from disjoint subsets equals the value for the union set:

$$S(T) = \sum_{\oplus} S(T_i), \; T = \bigcup T_i, \; \forall i \neq j : \; T_i \cap T_j = \emptyset. \tag{3.9}$$

This scheme of evaluation can be assigned to a CCG data structure in an obvious way by using CCG sub-trees as the subsets T_i. Thus, the evaluation for any CCG tree can be divided into sub-evaluations for every sub-tree. In order to generate the result the partial results have to be summarized and perhaps supplemented by some data from the sub-tree's root node. This scheme can be applied recursively.

For a single query on an uncompressed CCG this new scheme of evaluation provides no advantage in terms of computational effort. However, for successive queries as well as for compressed CCGs there is an opportunity for optimization. Whenever a sub-tree is to be evaluated multiple times, the result of the first evaluation can be cached and re-used for all following occasions, compare Figure 3.6.

For uncompressed CCGs this happens whenever a query is repeated for an actual sub-tree. Of course, it seems unnecessary to issue the same query more than one time. But if successive queries cover common sub-intervals of time, then all nodes and sub-trees thereof do become subject to re-evaluation. Very frequently there are queries where the former ones involve bigger time intervals A and the following ones consider only selected sub-intervals $B \subset A$ (zooming). Then, all sub-trees involved in the latter query have already been evaluated during the former. There is no need to re-calculate any intermediate results except for those nodes that intersect with the time intervals bounds, compare Section 4.5.3.

For compressed CCGs the same effect applies. Furthermore, nodes might be traversed several times during a single query operation because they are referenced more than once due to compression. Whenever an intermediate result for a node is required repeatedly, it can be taken from the cache saving the computational effort for re-evaluation. Thus, the reduction of computational effort equals the node compression, provided a full caching approach is applied. For reduced caching strategies with and further implementation details see Section 4.5.3.

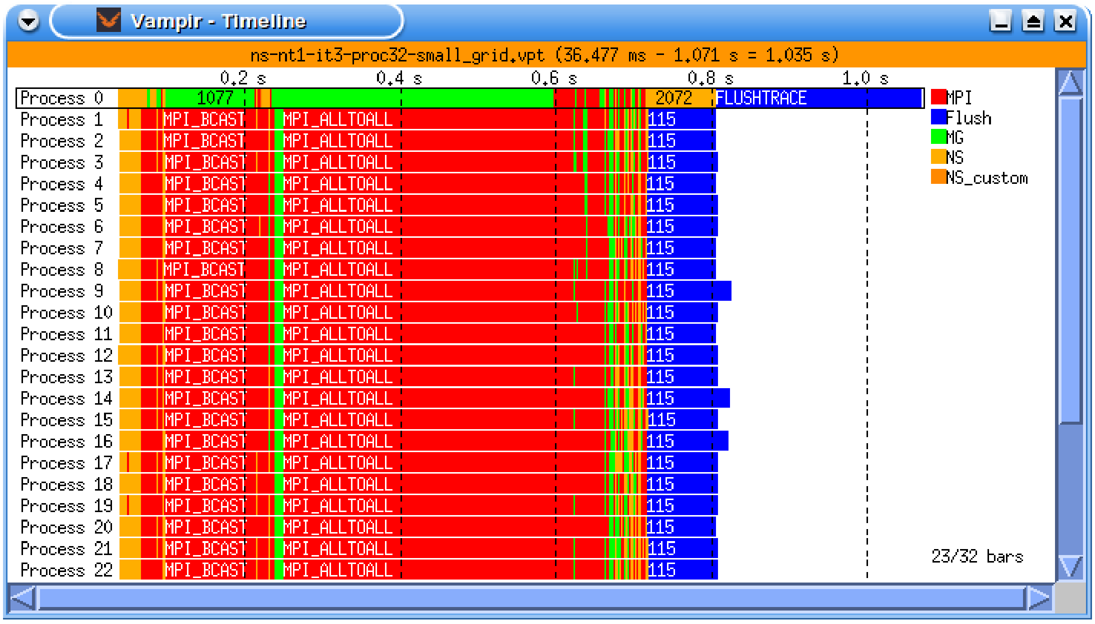


Figure 3.7: Vampir timeline display showing the function call behavior (by color) for a selected time interval (horizontal axis left to right) for a number of parallel processes (vertical axis).

3.4.4 The Timeline Query

Another important kind of request is deriving so called timeline visualizations. Basically, it arranges states (or single events) along a time axis to reflect the temporal behavior. Usually, the time is displayed along the horizontal axis from left to right. The vertical axis can be used to show different information, for example parallel processes or the call stack hierarchy of a single process, see Figures 3.7 and 3.8.

For a timeline visualization, the position and size of every colored box representing a program state must be computed. This could be done in a straight forward way mapping all states to corresponding pixel positions proportional to time with respect to a current view interval. This approach works well if the view interval contains rather few and rather long states. As soon as many states collapse into a single pixel this visualization method becomes unreliable. The usual solution for this problem is to select any of the coinciding states more or less by chance, compare Sections 2.2.1 and 2.4.1 and [NAW$^+$95].

Typical pixel resolutions for this type of visualization are in the range of today's display resolutions and can be considered almost constant. Practically, they range up to some thousand pixels in horizontal direction. In any case, it is several orders of magnitude less than the number of states inside any critical trace. The conventional rendering method, which is traversing all states or events of a trace, shows a linear computational complexity, proportional to the event count. Thus, this traditional algorithm is unsatisfactory with respect to correctness as well as with respect to computational effort.

A more expressive graphic representation should involve a sub-pixel aware rendering method. Whenever multiple states collapse into a single pixel, then the color information reflects this by composing colors appropriately. Figure 3.8 shows a comparison of the Vampir visualization and an alternative version using color blending.

Both, the traditional and the visually improved rendering algorithms can be deployed on compressed CCGs with unchanged computational effort (compare Section 3.4.1). Yet, the summary query introduced before can be re-used to compute the timeline diagram with less effort but in an improved manner with correct sub-pixel rendering. The general is to schedule a separate cached summary query for each pixel column. For every call depth level it has to collect all groups of functions (which are colored identically)

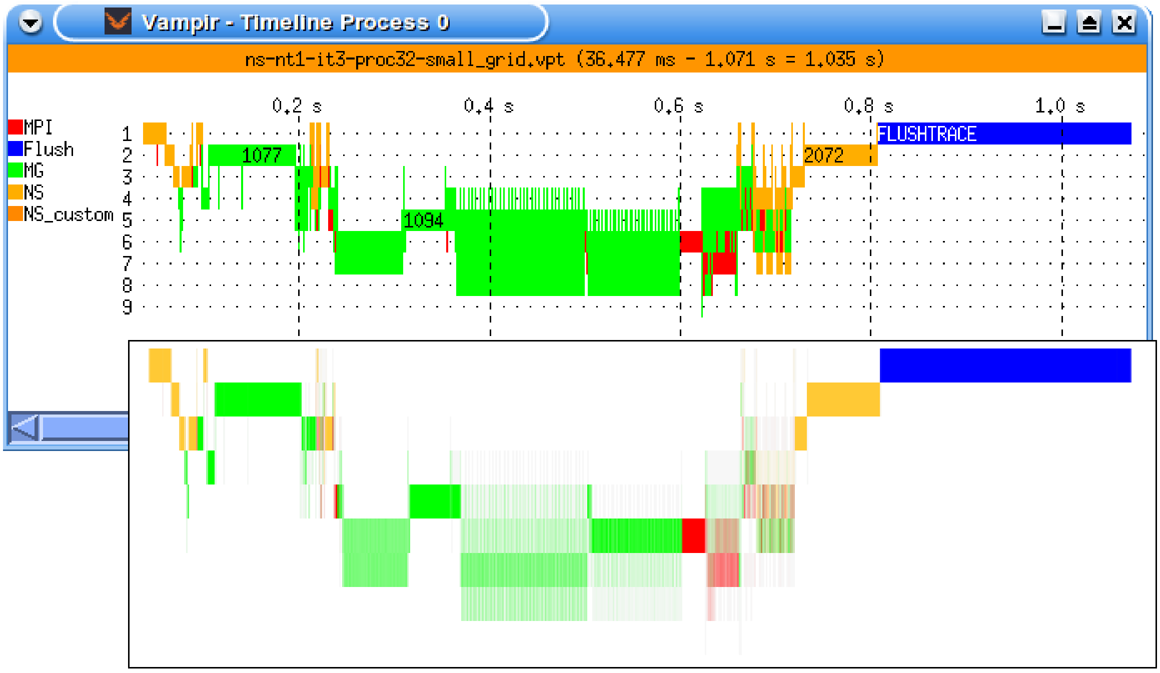


Figure 3.8: The Vampir process timeline display (top) in comparison with the improved timeline rendering output which uses color blending for a more authentic representation (bottom).

and their share of the run time. This information combined with the group-to-color mapping is sufficient to render the call timeline. Thus, the cached summary query can be used to reduce the computational effort for timeline visualization again. This underlines the importance of this kind of query. For further details, the algorithm and the complexity analysis see Section 4.5.4.

3.4.5 MPI Send-Receive Matching

For parallel programs following the point-to-point message-passing paradigm, a specific kind of query is required to match related message events. Usually, *send* and *receive* events are recorded separately within different processes. Therefore, a matching algorithm is required to compute the bijective relation of associated send and receive events, see Figure 3.9. This is essential for several purposes:

- determine causal relations between distributed events in parallel programs,
- measure message performance properties like duration and speed, and
- visualize messages between processes for human comprehension.

The mapping for MPI point-to-point messages follows the MPI Standard [For95, For97] which defines explicit rules for message matching:

- Messages match according to sender, receiver, communicator and tag parameters and according to temporal order.
- If sender, receiver, communicator and tag are equal for successive messages, then they must be delivered in same order as sent.
- If one of sender, receiver, communicator or tag differ, then messages need to be delivered as constrained by the receiver.
- Data type and message length are not taken into account for matching.

Wildcard parameters (like `MPI_ANY_SOURCE`, `MPI_ANY_TAG`) need to be resolved during tracing already, the measurement system needs to determine the actual values. Thus wildcard parameters will

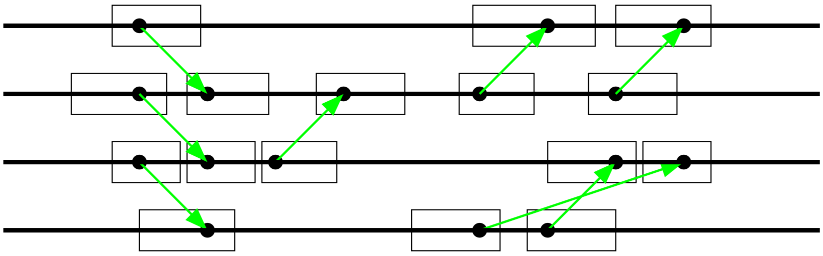


Figure 3.9: Matching of send and receive events on parallel processes according to the MPI standard. The visualization connects associated send and receive events with arrows.

not be encountered in trace analysis. Relying on this rules, the matching algorithm will be MPI specific. For other message passing standards there should be other well-defined rules that allow a corresponding matching scheme.

According to above-mentioned rules, a matching algorithm cannot work on individual message events. It rather needs to include all preceding message events. Therefore the match algorithm needs to traverse all participating process traces from the beginning on in temporal order. This corresponds to a form of partial de-compression when applied to the compressed CCG data structure. Therefore, the computational complexity for send-receive-matching is at least $O(N)$ for N send or receive events, even when applied to a CCG representation with $n < N$ nodes. See Section 4.5.5 for the detailed algorithm description and Section 5.5 for evaluation.

3.4.6 MPI Collective Operation Matching

Matching of collective communication events in MPI follows the same principle as for point-to-point message events in Section 3.4.5. Again, there are separately recorded events within all participating processes which are connected by the mapping operation, compare Figure 3.10.

According to the MPI Standard [For95, For97] the mapping algorithm can act very similar to the one for point-to-point messages (see Section 3.4.5). However, one needs to consider fewer properties:

- The communicator which the collective operation is in.
- The strict temporal order within each communicator.

As only one collective operation can be active on any process at any time, the type and all remaining parameters of related collective operation events must always match for accurate traces. Matching of MPI collective operations is considered as a special case of send-receive matching and is not explicitly referred to further on.

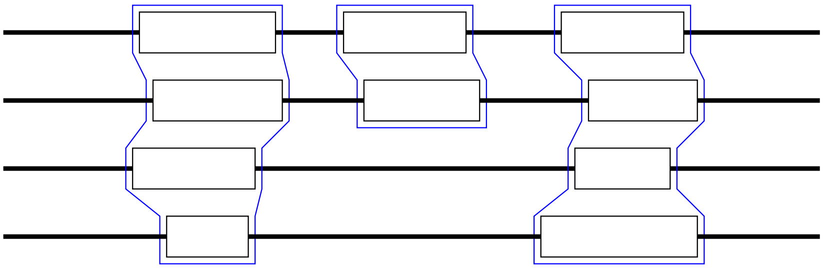


Figure 3.10: Matching of n collective MPI communication events to a common parallel event.

3.4.7 Automatic Analysis Methods

Besides the fundamental trace evaluation, a variety of methods for *automatic trace analysis* is known in literature [WM99, WM00c, FGS03, WMDM07]. Automatic analysis aims to provide readily interpretation of trace data including a rating of the performance properties. It relieves the human user from the task of searching for critical spots due to frequently found performance problems.

Usually, such critical spots do not consist of single events but of *situations*, i.e. conglomerates of related events. This may comprise for example certain function call sequences (local events only) or message interchange (local and remote events). Semantic information about certain types of situations is used to estimate the optimal way of processing and the anticipated performance. Finally, the situations are reported together with qualitative or quantitative rating. The former classification divides situations into *adequate* and *critical* with respect to performance. The latter gives a scalar weighting or a severity value in addition. Both allow to select only the critical ones respectively the most severe ones for presentation to the user [WM98, WM00a, GMT04].

Even though automatic trace analysis covers many different *performance situations*, it only uses few basic building blocks for accessing the underlying trace events [WM00b, FGM$^+$01]:

- searching for time stamps and for certain events or situations,
- iterating over events of processes,
- accessing single events and their properties,
- accumulating statistics about events and their properties, and
- relating events locally and remotely.

All of these tasks are covered by the fundamental query algorithms presented above, compare Sections 3.4.2 to 3.4.6. Therefore, existing and future algorithms for automatic analysis of trace data will be able to rely on the functionality provided by the CCG data structure.

3.5 Persistent Storage and Restoring

So far, the general conception of trace analysis based on compressed CCGs is to create the data structure on demand, i.e. the compressed CCG is created when reading a trace from a file. Then any desired analysis is performed on the memory data structure which is dismissed at the end of an analysis session.

Obviously, this causes unnecessary overhead when a trace is read repeatedly. It could easily be avoided by saving the compressed CCG persistently and restoring it later with much less effort, see Section 4.6. This process is also known as *serialization*, because an arbitrary data structure is mapped to a medium that is supposed to be read and written in a serial manner like a file or a network stream [Eck95].

By this means, a trace file compression scheme can be derived as a by-product of the CCG data structure. This is suitable for archiving and data transfer. It is sensitive to all compression parameters like error bounds and branching factors, because it carries altered information like the associated compressed CCG does. Therefore, it is to be regarded as a lossy compression scheme as well. Depending on the encoding of the serialized data structure, it is suggested to combine the serialization of compressed CCGs with any block compression scheme, like *gzip* or *bzip2*. This allows the application of a simple, robust and human readable encoding in plain ASCII text without disadvantages for resulting file sizes.

Beyond the scope of *in-memory* compression, there are some ideas for further improving *trace file* compression. First of all, file compression always requires a decompression operation. This is an evident difference to the transparent way compressed CCGs can be accessed. As soon as there is a decompression phase, additional techniques can be applied in order to improve compression. In this case, most notably this is differential coding. It relies on the fact that a graph node can be specified by giving a reasonable similar representative node and indicate all differences by few correction terms. With respect to storage size, the pair of representative node and correction terms might be shorter than the separately stored nodes. On one hand, differential coding could be used to create a lossless compression method were all differences in similar nodes are appended as correction terms. On the other hand, lossy compression could be enhanced even more by further compressing the remaining representative nodes.

There is a most notable difference when comparing the compressed CCG data structure and the derived trace file compression method. For compressed CCGs in main memory, rather small branching factors are recommended. Contrary to this, trace file compression with differential coding prefers large branching factors because of the improved ratio of the size of a complete node and the size of the correction terms. See [Knü03] for further information.

By itself, the compressed CCG approach seems too expensive to be used as a sole compression and decompression tool for trace files. The computational effort and the main-memory requirements are quite high, in particular when compared with block compression algorithms like Lempel-Ziv [ZL77, Wel84] and others or tools like *gzip* or *bzip2*. Yet, it can provide higher compression ratios than general purpose compression algorithms. This applies to lossless compression as well as to compression with very narrow deviation bounds and even more so to compression with substantial deviation bounds.

The main purpose for the persistent storage is to reduce the delay at the beginning of an analysis session. Construction and compression of very large traces on a workstation might be quite time consuming. Compared with this, restoring of a previously compressed CCG is much quicker. The initial delay is reduced from several minutes to some seconds.

As a solution the actual construction and compression with moderate compression parameters should be scheduled right after tracing time on the original HPC platform. It could either process all merged process traces together or every process trace on its own. It is not suggested to perform CCG compression during tracing time! The overhead in computational effort and memory consumption would entirely disturb the observation of the program execution.

3.6 Summary

The presented combination of the CCG data structure with inherent compressibility and the associated query algorithms promise a general advancement in trace analysis. On one hand, it reduces the memory footprint of the data itself. This allows to analyze larger traces that would be inaccessible otherwise. On the other hand, it decreases the computational effort of typical query algorithms. The acceleration of analysis is particularly important for interactive analysis and visualization tools.

The following Chapters 4 and 5 will focus on detailed algorithm design and theoretical assessment respectively on experimental validation with real-world examples.

4 Algorithms for the CCG Data Structure

This chapter is dedicated to the algorithms for construction, compression and evaluation of the newly introduced data structure. It will provide implementation details for the ideas from the previous chapter and investigate their computational complexity.

4.1 CCG Construction

For optimal efficiency the construction and compression algorithms for Complete Call Graphs are closely incorporated such that at no time an intermediate uncompressed CCG needs to be stored. For the sake of explanation they are discussed separately in the current and the next section. Section 4.3 shows how both are combined.

4.1.1 General Construction

The construction of a CCG from a program trace relies on a single read-through of all recorded events in temporal order. Without loss of generality the CCG construction focuses on a single process trace only. For parallel traces the CCG construction can be performed in parallel but independently, since there are no inter-process relationships at this stage.

There are three groups of events to distinguish in the construction process: function *enter* events, function *leave* events and *atomic* events. The former two are mapped to *call nodes* which represent function calls within a trace. A call node carries a function identifier and a list of $n \leq b$ child node references (pointers). Furthermore, it represents $2 \cdot n + 2$ time stamps, including its own *enter* and *leave* time stamps as well as the *enter* and *leave* time stamps for all child nodes. During this early phase of node creation, timing information is actually stored as time stamps, not yet as duration values.

For all *enter* and *leave* time stamps the following Lemma applies:

Lemma 2 (Stack Property). *Let $\sim$ be the one-to-one relation that assigns every enter event e to its associated leave event $l \sim e$ and vice versa. Then the enter event precedes its associated leave events in terms of time:*

$$e < l. \tag{4.1}$$

For two pairs of associated events $e_A \sim l_A$ and $e_B \sim l_B$ in the same process there is

$$e_A < e_B < l_A \Longleftrightarrow e_A < l_B < l_A. \tag{4.2}$$

Proof. From the procedural programming paradigm follows that a function call ends after it has been started which implies (4.1). The stack property (4.2) is directly inherited from the procedural programming model which defines a LIFO scheme for function calls. □

Atomic events cover all remaining kinds of events. They are always leaf nodes, i.e. have no children, and carry specific properties according to their specific types. Nodes representing *atomic* event, which relate to a single time stamp, carry no timing information. Instead, the time is included by the direct parent node which necessarily is a call node.

The CCG creation procedure starts with an initial empty call node with start time 0 which is going to be the root of the tree. At the same time, this is the initial active node. Now, for every event an appropriate modification to the existing tree is performed in order to insert the associated information into the CCG:

Enter events: For every *enter* event e create a new function call node F with start time t_e and function identifier taken from e. Furthermore, append F as latest child node to the current active node P and append t_e to the time stamp list of P. Finally, F becomes the new active node which is left pending until the respective *leave* event $l \sim e$ arrives.

Leave events: For a *leave* event l the current active node F must have been created by the associated *enter* event $e \sim l$ because of the stack property (4.2) which can be checked by comparing the function identifiers of l and F. Then, append the time stamp t_l to the time stamp lists of the current node F and its parent node P. Finally, make P the current active node again and finalize node F.

Atomic events: For *atomic* events a separate node is created that does not depend on any earlier or later events. Therefore, node creation and finalization can be performed in one step.

Finalization: On finalization of a node it is guaranteed that all child nodes have been finalized before according to (4.2). At first, the time representation is transformed from $2 \cdot n + 2$ time stamps t_i to $2 \cdot n + 1$ time durations d_i:

$$d_i = t_{i+1} - t_i, \ i = 0, ..., 2n. \tag{4.3}$$

Then, the actual CCG compression looks for a replacement for the current node (see Section 4.2). If successful, the node is deleted. Otherwise, the node is copied to a permanent representation with an optimized encoding (see Section 4.1.3). In either case, the reference from the parent node P to the current node is updated to point to the correct new represenataion.

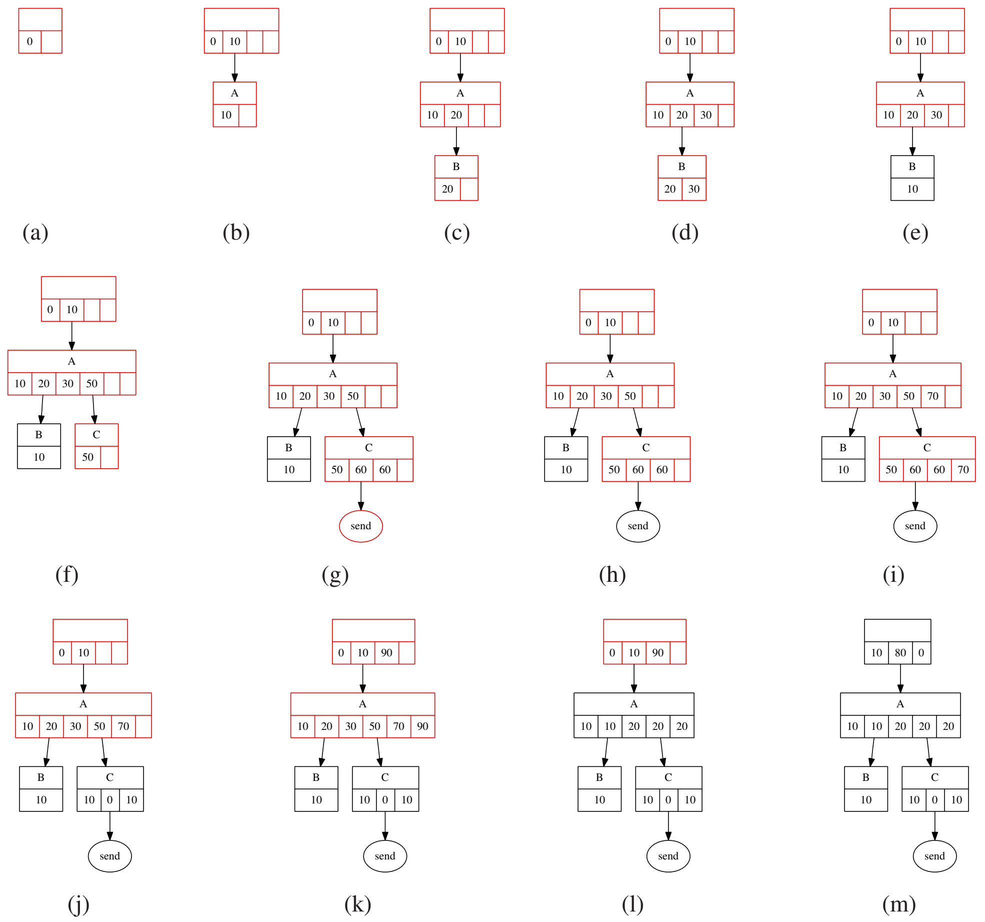
In Figure 4.1 the CCG construction process is demonstrated for the following example trace:

```
10 enter A
20 enter B
30 leave B
50 enter C
60 send
70 leave C
90 leave A
```

The computational complexity for CCG construction without compression as described here is $O(N)$ for N events. For every event, either node creation or node finalization or both have to be performed, causing constant effort per event. The time transformation has constant effort per event, too, because every time stamp delivered by an event takes part in not more than three subtractions: one inside the current node and two in the parent node.

4.1.2 Splitting of Wide Nodes

According to Section 3.2.2, wide nodes with child count $n > b$ have to be split into a tree of artificial nodes. A straight forward implementation would wait until finalization of a wide node and derive a balanced tree of artificial nodes. In general, this approach works fine. In particular, a balanced tree would always have minimum depth $\log_b n$. However, this way of splitting has a severe drawback concerning very wide nodes with $n \gg b$ children. Then, the complete node would have to be stored before it can be transformed. Only after this, the compression scheme can reduce the memory requirements. Thus, this simple splitting approach has temporary memory requirements of $O(n)$.



(a) (b) (c) (d) (e)

(f) (g) (h) (i)

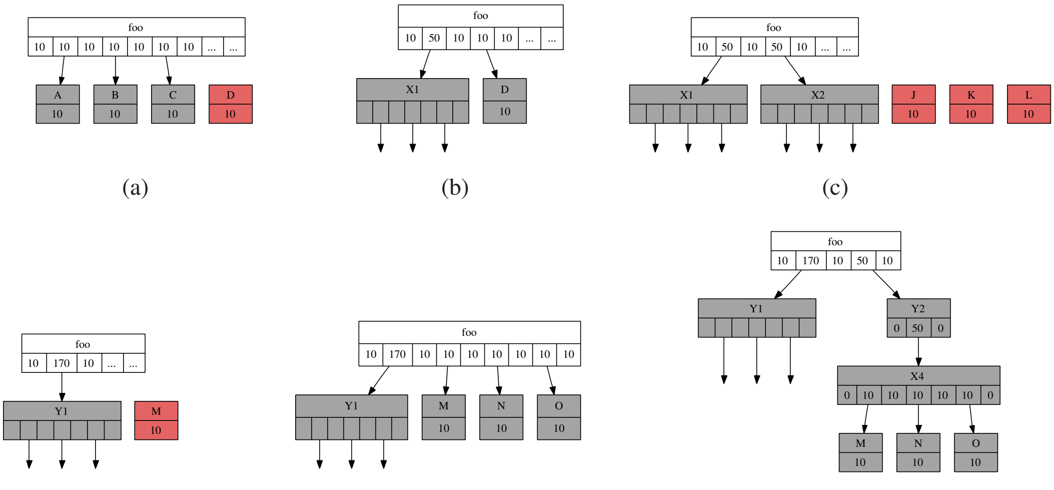
(j) (k) (l) (m)

Figure 4.1: Demonstration of CCG construction: Starting with an empty root node (a) the function A is entered at time 10 (b). Then sub-function B is called at time 20 (c) and immediately left at time 30 (d). Finalization of node B results in (e). A call to function C at time 50 creates a corresponding node (f). Then a *atomic* event $send$ at time 60 creates a leaf node (g) which is finalized instantly (h). Now function C is left at time 70 (i) and finalized (j). Finally, function A is left at time 90 producing state (k) and in turn state (l) after finalization. As the end of the trace is reached there is still the root node to be finalized. Thus, (m) is the resulting CCG.

In particular, for degenerated flat traces this would be unacceptable. Such traces consist only of a main function with direct child function calls but no deep call hierarchy. While inappropriate for a naive CCG implementation, such traces are perfectly valid originating for example from limited instrumentation where only MPI communication functions are considered which contain no child function calls.

A more advanced algorithm allows to create the artificial nodes while new children are appended to a wide node F and before the finalization of F. The algorithm works in two stages: An *early stage* is performed just before the next child node is to be appended. This stage limits the actual number of child references to $\leq b \cdot \log n$ instead of $\leq b$. This is a sufficient reduction for practical purposes compared to the original number n. Finally, the *late stage* enforces the strict limit of b children per node. See Figure 4.3 for an example.

During early stage splitting, all nodes are assigned a *child level*. This is necessary in order to create a tree of artificial nodes with minimum depth and branching factor b (*minimum-depth tree*). All newly added child nodes get the initial level 0. Further on, groups of children of same level are turned into a new artificial node, whose level is therefore incremented by 1. Before a new child is appended to a node there



(a) (b) (c)

(d) (e) (f)

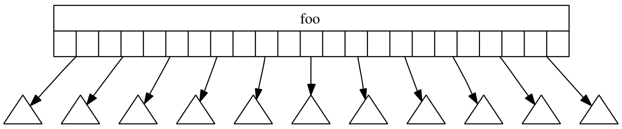
Figure 4.2: Example of on-the-fly splitting of wide nodes. In (a) there is a function `foo` with currently three children `A`, `B` and `C` which have already been finalized and may have been replaced already (gray). A fourth child `D` is about to be appended. With the limit $b = 3$ the child level 0 of node `foo` is filled up. Before `D` can be added, early stage splitting takes place. All current children `A`, `B`, `C` are replaced by an artificial node `X1` of child level 1. Then `D` is added in child level 0 (b). Both, `X1` and `D` are subject to finalization and compression at once (gray). After some more children have been added there are two artificial nodes `X1` and `X2` of child level 1 present and another three children `J`, `K`, `L` are waiting to be added (c). In turn `J`, `K`, `L` are replaced by level-1 node `X3`. Now, the three level-1 nodes `X1`, `X2`, `X3` are transformed to a new artificial node `Y1` of level 2 (d). Three more children `M`, `N`, `O` are appended before finalization of node `foo` (e). Now `foo` has $4 > b$ child nodes. Prior to finalization late stage splitting will reduce this number to $\leq b$ by introducing `X4` of level 1 and `Y2` of level 2 (f). Compare Figure 4.3 for the complete artificial sub-tree generated from wide node `foo`.

is a check, if the lowest level l already contains b entries $L_1, ...L_b$. If so, those entries are replaced with a new *artificial node* A which adopts the children L_i. This reduces the current child count of node F by $b - 1$. The artificial node A is inserted to the next higher child level $l + 1$ and the procedure is iterated until a level with $< b$ entries is reached. Then A can be finalized and replaced immediately, following the normal course of the compression scheme, compare Figure 4.2. By this means, artificial nodes are created, finalized and compressed while the parent node F itself is still growing. This works without prior knowledge of the final child count of a wide node.

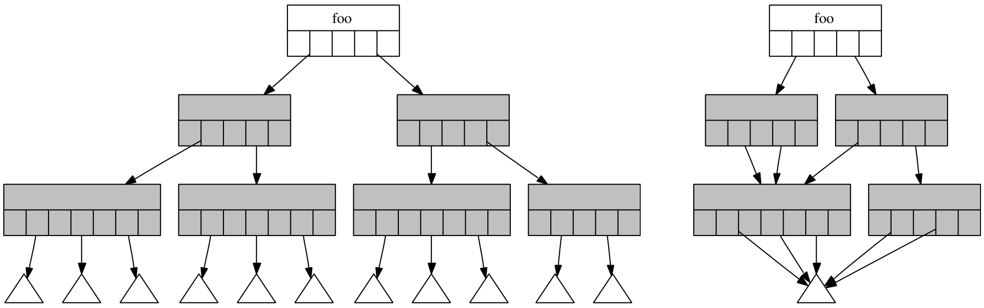
Complementing the *early stage* of node splitting there is the *late stage* performed just before node finalization. It has to terminate the intermediate states of the $l = \log_b n$ pending child levels with $m_i \leq b$ entries each. For every pending child level l an additional artificial node is generated and inserted to the level $l + 1$. This works just like in early stage but regardless of the number of entries per level.

Figure 4.3 shows a comparison of the resulting unbalanced *minimum-depth tree* with its strictly balanced counterpart. Both are guaranteed to have the same minimal depth, which is the most important feature. Therefore, search operations in either one have optimal logarithmic complexity. Yet, the temporary memory requirements are drastically reduced from $O(n)$ for the balanced tree to $O(b \cdot \log n)$ for the *minimum-depth tree*.

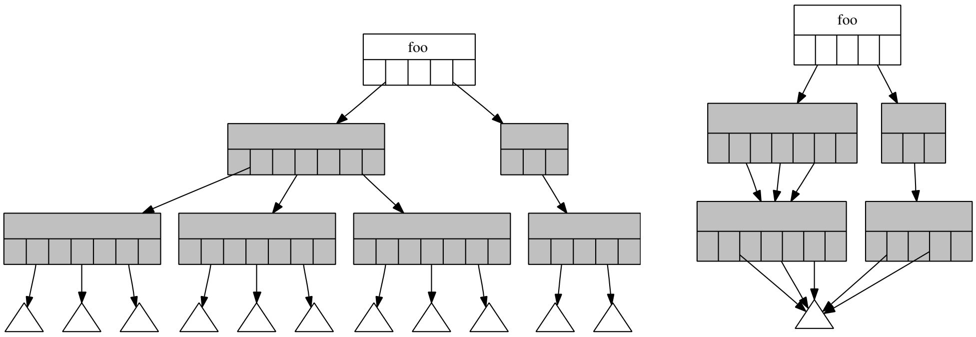
Furthermore, the minimum-depth approach splitting has an advantage in terms of the following node compression. While the balanced splitting would create an even spectrum of artificial nodes with $n = 2...b$ children per node, the minimum-depth splitting creates the majority of artificial nodes with b chil-



(a) Node `main` with 11 children without splitting,



(b) ... after balanced splitting (left uncompressed, right compressed),



(c) ... and after minimum-depth splitting (left uncompressed, right compressed).

Figure 4.3: Comparison of a wide CCG node `main` without splitting (a), after balanced node splitting (b) and after the minimum-depth splitting (c). For a detailed explanation of the early and late stages of splitting see Figure 4.2.

dren during the early stage plus a few remaining ones with $\leq b$ children duing the late stage. Under optimal conditions, all artificial nodes on the same child level and with the same child count are compatible to one another. Then, the minimum-depth splitting scheme allows better compression, see Figure 4.3 for an example.

4.1.3 Graph Node Encoding

The graph nodes representing trace events are regarded as *atomic* items for the CCG approach. This section focuses on the encoding of the nodes.

In the life cycle of every CCG node there are two phases, that are covered by two different node data types. Before node finalization and comression, CCG nodes are stored in temporary construction data types. During node finalization, CCG nodes are either removed due to compression or transformed into a permanent representation.

The construction nodes a stored in a generic data type, that is able to hold any node type as well as additional properties that are only required until finalization. This allows to fill in partial information before the actual node type is determined. The memory consumption of construction nodes is not critical, because there is only a limited number of at any point in time.

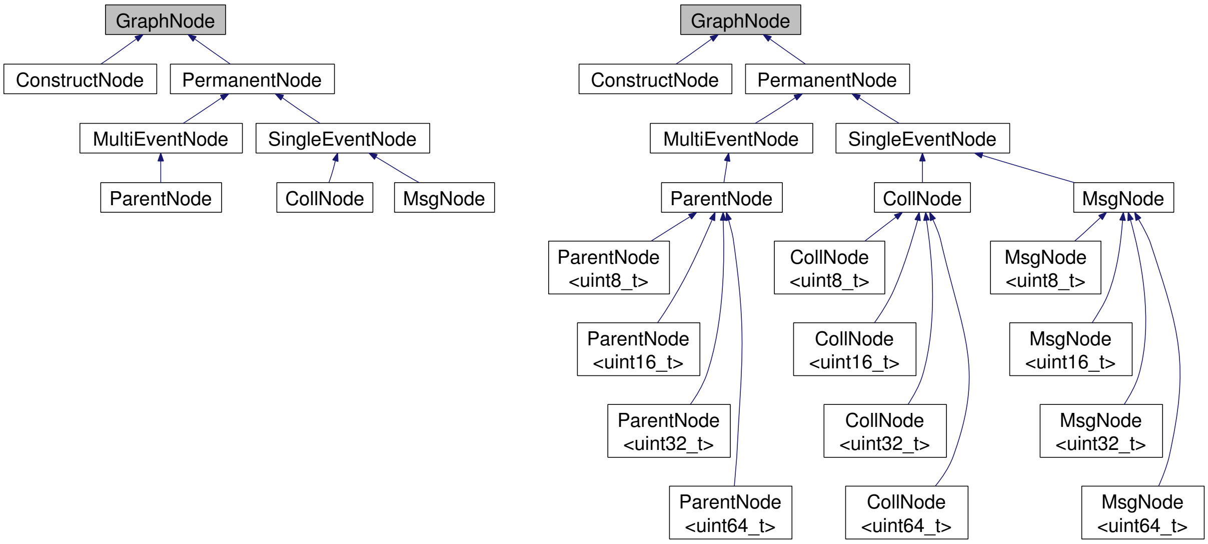


Figure 4.4: OOP class hierarchy for node data structures (left). `ConstructNode` represents any node type prior to finalization. Later nodes are copied to an instance of the non-abstract sub-classes of `PermanentNode`: either `ParentNode` (for function calls), `CollNode` (for collective MPI communication events) or `MsgNode` (for point-to-point communication events). The hierarchy is extended by template meta-programming (TMP) in order to reduce over-all memory consumption (right).

The permanent node are stored in separate compound data types for different node types. They share some common properties which should be accessible in a uniform manner, but also have some type specific properties. Furthermore, permanent nodes are read-only data structures and are critical with respect to memory consumption, because they constitute the actual CCG. Therefore, the data types for the different node types are modeled in an OOP (object oriented programming) class hierarchy, featuring convenient virtual access methods for common node properties, see Figure 4.4(left).

With this storage scheme for permanent nodes, the total memory consumption of CCGs is approximately twofold compared to sequential data structures (arrays). This is mainly due to the additional references (pointers) between CCG nodes, that are not present in sequential arrays.

Yet, the CCG approach allows further optimization of the node storage scheme: The time duration values contained in the call nodes are usually much smaller than the original time stamp values. Therefore, it is rarely necessary to use a universal 64 bit integer variable. Instead integer variables of 32 bit, 16 bit or even 8 bit length would be sufficient often[1].

Therefore, the OOP class hierarchy can be extended to include node types with different integer types for time durations, see Figure 4.4(right). On creation of every permanent node the smallest integer type is selected, that is sufficient to hold all duration values.

This scheme is transparent to read access via virtual methods, yet, node finalization and transformation become more complicated. In particular, this has negative impact on the code size and its maintainability. However, this concept can be put into practice elegantly and very efficiently by using C++ *template meta programming* [Vel95a, Vel95b]. Templates are a C++ language concept providing parameterized data types and function definitions. They are evaluated at compile time, not at run time. Templates are fundamentally different from objects and inheritance between oriented classes, but it can be combined with them very nicely. For more information on this see relevant literature about C++ in general and C++ templates in particular [Eck95, Vel95a, Vel95b].

---

[1] An alternative solution could be run-length encoding for the time duration values in order to remove leading zeros.

4.1.4 Graph Node Allocation

All nodes taht are not replaced during CCG construction need to be stored permanently. This results in (potentially) large memory consumption which is accumulated by many small pieces. The typical size of a single permanent node is 20 to 200 bytes. Yet, the total memory consumption may as well grow to the order of magnitude of the available main memory size.

The memory is allocated via the memory management interface of the operating system, for example via the POSIX `malloc()` function. If implemented in a naive way, this would imposes cause notable overhead with respect to run-time, because of the frequent memory allocation calls, and with respect to memory consumption, because of internal memory management information per allocated block [BMBW00].

Frequent calls to the memory management interface can be avoided by introducing an adapted memory allocation scheme. It uses two different approaches for construction time nodes, which are frequently created and deleted, and for permanent nodes, which are created consecutively without intermediate deletion.

At any point in time, there is only a small number of construction nodes present, thus, they will not become critical with respect to memory consumption. Their number corresponds to the current depth of the function call stack. Therefore, construction nodes are arranged in a stack data structure where they are re-used, diminishing almost all memory allocation operations on their behalf.

The permanent nodes account for the major memory consumption of any CCG. They are created consequtively and with out intermediate deletion or re-allocation. Therefore, permanent nodes can be placed at consecutive locations inside a large pre-allocated memory area. By this means, many small allocations are replaced by a few large ones eliminating the run-time overhead as well as the memory consumption overhead almost completely.

4.2 CCG Compression

The compression of a CCG is achieved by replacing maximum compatible sub-trees with references to a single instance. Instead of comparing whole sub-trees in full depth it is desirable to use a comparison scheme with constant complexity, independent of the size or the depth of the sub-tree. This is possible given that all nodes have no more than $b = constant$ children (see Sections 3.2.2 and 3.3.2).

The pairwise sub-tree comparison considers only the top nodes of every sub-tree and its child references (child pointers). If the top nodes match with respect to node type, hard properties, and soft properties and all pairwise corresponding child references point to the same sub-tree, then the complete sub-trees are guaranteed to be compatible. This means, the test for identity of arbitrarily large sub-trees is reduced to a single pointer comparison.

This simplification is only valid, if corresponding compatible child nodes have been compared and re-referenced to the same replacement node before, see Section 3.3.2. Therefore, this optimized scheme determines the order in which the graph nodes have to be handled to a bottom-up order from child nodes to their parent nodes.

Fortunately, this is the very order of node finalization in the course of the construction process, see Section 4.1. Thus, compression can be incorporated into the finalization step, such that compression for a single node is performed right after its creation. See Figure 4.5 for an illustrative example.

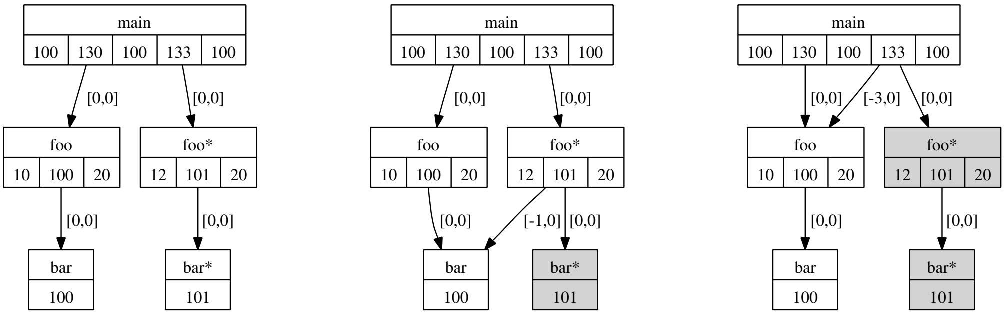


Figure 4.5: Successive compression in a CCG. The first graph (a) shows the uncompressed sub-tree. After the finalization of node `bar*` it is found to be compatible with node `bar` and replaced by a reference (b). Note the slight difference in run-times: the deviation arising from this is propagated to the parent node as a deviation interval `[-1,0]`. After finalizing `foo*` it is replaced by a reference to `foo` as both reference the same child node (c).

4.2.1 Search for Replacement Nodes

In the course of CCG compression, for every node (sub-tree) C a search for a compatible replacement node (sub-tree) P is performed, see Figure 4.5. The search operation considers the following node attributes:

1. node type,
2. child references (pointers),
3. hard properties, and
4. soft properties.

The optimized search method for replacement nodes uses the first three node attributes in order to quickly locate a sub-set of replacement candidates. Only those are subject to the expensive one-by-one comparison with respect to soft properties. For the node C a hash value $h = h(C)$ is computed from the *node type*, the *child node pointers* and its *hard properties*. Then h is used as index in a hash data structure containing all previous nodes that are not eliminated in the course of compression.

The hash data structure contains N_{hash} bins for all values $(h \mod N_{hash}) \in [0, N_{hash} - 1]$, compare Figure 4.6. To every hash bin there is a *collision list* which distinguishes nodes with identical hash values but different attributes. To each collision list entry there is a so called *S-list* of similar nodes whose entries differ only in soft properties. The collisions lists are usually rather short, provided there is a reasonable hash function [JJ97] and the number of all *S-lists* in the whole data structure is in the same order of magnitude as N_{hash}. The latter can be achieved by adaptive re-hashing as the lengths of the collision lists exceed a certain threshold.

Finally, the soft properties (fourth attribute) are examined for all replacement candidates in the *S-list*. Each entry S_i is compared to C with respect to all soft properties, see Section 4.2.4. The first matching entry is selected as replacement for C and the search is aborted. If no entry is found, then C is appended to the end of the S-list where it is available as future replacement candidate.

4.2.2 Caching of Nodes

In general, the length L of the S-list S is unbounded and effort for linear traversal is $O(L)$. In order to achieve constant over-all effort for the search operation, L can be restricted by a constant bound such that only the L^* most recent candidates are respected. The limited search method is implemented with a

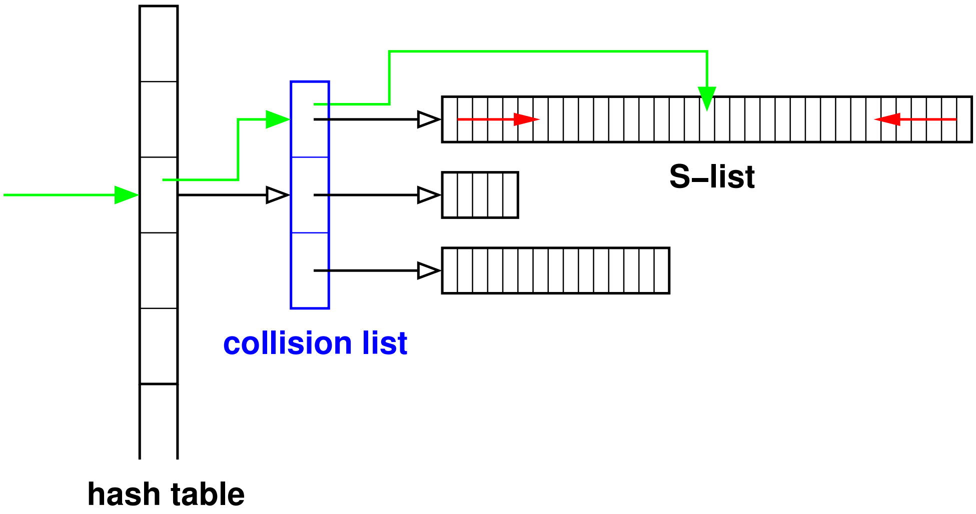


Figure 4.6: Hash data structure for graph nodes. Each hash table entry points to a collision list (blue). Every collision list references multiple S-lists, which contain all nodes with common hard properties. There are various strategies for searching within S-lists, see below.

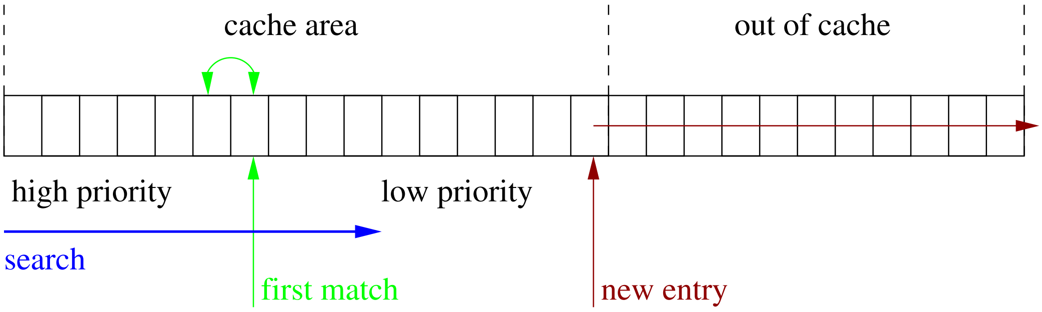


Figure 4.7: Search and caching scheme within S-Lists: There are $\geq M*$ entries in the *cache area* which are considered in search operations in strict front-to-back order(blue). The first match is taken as a result. Furthermore, this element is swapped with its predecessor in order to increase its priority (green). If no match is found then a new entry is inserted at the end of the cache area, pushing the previous item at this position out of cache (red).

priority cache with the *least frequently used* strategy (LFU). Every S-list (compare above) is managed as a separate cache, such that only the first L^* entries are actually regarded as cache area and later entries are ignored, but still kept in the storage container, compare Figure 4.6.

Searching is performed in a front-to-back manner and stops at the first actual match, the corresponding item is used as result. As a side effect, the result item at index i is swapped with its predecessor at index $i-1$, $(i > 2)$. Though this means, priority is increased for successfully matching entries.

If the search turns out negative, because there is no matching item among the first L^* entries, then a new one is created at (or near) the end of the cache. This gives minimum priority to newly created entries. The previous entry at this position is moved out of the cache area to the end of the S-list, compare also Figure 4.7.

The run-time effort of this caching scheme still grows linear with the length of the S-list like for full search. Yet, it is bounded by a constant L^*. The additional effort for the priority cache is constant per search operation.

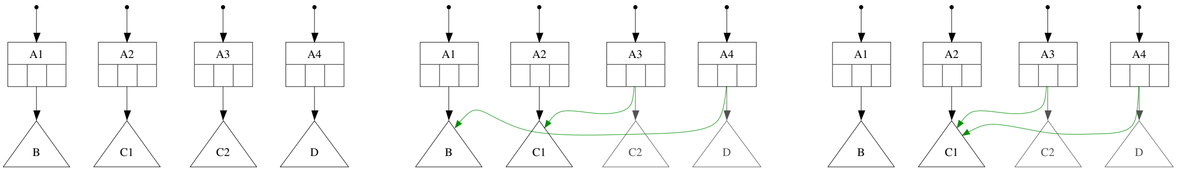


Figure 4.8: Effect of search order: assume four sub-graphs that are identical in terms of hard properties (left). Nodes $A1$, $A2$, $A3$, $A4$ are pairwise compatible (i.e. matching soft and hard properties). $C1$, $C2$ and D are pairwise compatible, too. B is compatible to D but not to $C1$ and $C2$ (not transitive). With front-to-back search without re-ordering the compression results in the configuration where D is replaced by B (middle). With the priority cache scheme introduced above, D is replaced by $C1$ (right). This influences the compression for the parent node $A4$: it can either be replaced only by $A1$ (middle) or by $A2$ and $A3$ (right).

4.2.3 Influence of the Node Search Order

The search for replacement nodes always returns the first compatible node instead of the *best* match (according to any rating), because of the tremendous computational complexity for an optimal selection.

Therefore, the search order influences the compression result. In particular, if there are multiple valid replacements for a candidate node, then the first one tested will win. For compatible parent nodes it is crucial, that each pair of corresponding child nodes is replaced by the same reference (if compatible), see Section 3.3.2. Otherwise, the parent nodes cannot be re-referenced to the same instance. Because of this, a closer look at the effects of the search order to the compression is necessary. Three search strategies are discussed below:

- front-to-back search,
- back-to-front search, and
- priority cache search (according to Section 4.2.2).

The front-to-back search always tests nodes in the order of creation, i.e. it returns the earliest compatible node. In case the earliest replacement is only able to cover a small share of all nodes (of a group) and a more recent node is covering a larger share, then the latter cannot be preferred over the earlier. This is a severe disadvantage in the long run. Furthermore, the front-to-back search is unsuitable for a limited search approach. If only the L^* earliest nodes are regarded, no new nodes can be incorporated once there are more than L^* entries which could effectively disable any further compression.

Unlike front-to-back search, the back-to-front strategy is more suitable for limited search. It always picks the most recent compatible replacement, thus it will adapt to new nodes very quickly. Unfortunately, this conflicts with the requirement, that corresponding child nodes should be replaced with the same references. If new nodes appear in between the compression of the first and the second set of child nodes, then the new nodes are preferred over the previous ones. As a consequence, sporadic new nodes at leaf level will destroy an established compression path for the parent node, it's parent nodes and so on.

The priority cache search (see Section 4.2.2) is adaptable to new situations and at the same time regular enough to ignore occasional disturbances due to newly added nodes. It increases the most frequently used node within a limited part of the compression history (depending on L^*) inserts new nodes with low priority. Thus, short-term changes are ignored while long-term changes are taken into account.

As a positive side effect, the latter scheme reduces the number of node comparisons. It increases the priority for frequently referenced nodes and thus decreases the number of comparisons before this node is found the next time. On average, the effort in the more probable cases is decreased at the expense of the less probable cases, reducing the over-all effort.

Figure 4.8 shows an example for *front-to-back search* and *priority cache search*. Assume, that in the original situation in Figure 4.8(left) the nodes $C1$, $C2$ and D are pairwise compatible and B is compatible to D but not to $C1$ and $C2$. With *front-to-back search* D is always re-referenced to B – Figure 4.8(middle). Using *priority cache search* the replacement chosen depends on how often it has been used before. In the given scenario, $C1$ is picked instead of B, because B is used only once (for itself) whereas $C1$ is used twice.

4.2.4 Node Comparison

The comparison of two nodes with respect to all soft properties tests if the difference of corresponding values lies within the allowed deviation bounds. Sometimes, the deviation bounds are defined globally but in some cases it needs to be adaptive. In particular, the absolute time deviation for nodes needs to be propagated from children to parent nodes as introduced in Section 3.3.2. The actual deviation estimation for sub-trees is performed with deviation intervals[2] instead of absolute values. This improves compression notably, because it allows mutual cancellation of positive and negative deviations.

Lemma 3. *Interval arithmetic performs operations on intervals* $a = [a_*, a^*] \in \mathbb{IR}$ *instead of scalar numbers* $x \in \mathbb{R}$. *All intervals have to be non-empty, i.e.* $a_* \leq a^*$. *Point intervals* $a = [x, x], x \in \mathbb{R}$ *are allowed. Monadic and dyadic operations* $F : \mathbb{IR} \to \mathbb{IR}$ *(e.g.* $abs, sin, cos, ...$*) and* $\odot : \mathbb{IR} \times \mathbb{IR} \to \mathbb{IR}$ *(e.g.* $+, -, *, /, ...$*) based on corresponding scalar operations* $f : \mathbb{R} \to \mathbb{R}$ *and* $\circ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *can be defined in a general way as:*

$$
\begin{aligned}
F(a) = F([a_*, a^*]) &:= \left[\min_{x \in a}(f(x)), \max_{x \in a}(f(x))\right], && (4.4)\\
a \odot b = [a_*, a^*] \odot [b_*, b^*] &:= \left[\min_{x \in a}\min_{y \in b}(x \circ y), \max_{x \in a}\max_{y \in b}(x \circ y)\right]. && (4.5)
\end{aligned}
$$

Following this definition, interval addition $+$ *and interval subtraction* $-$ *are no longer inverse operations [AH83]. The alternative operation* $\ominus$ *can be introduced as inverse interval addition:*

$$
\begin{aligned}
[a_*, a^*] + [b_*, b^*] &:= [a_* + b_*, a^* + b^*] && (4.6)\\
[a_*, a^*] - [b_*, b^*] &:= [a_* - b^*, a^* - b_*] && (4.7)\\
[a_*, a^*] \ominus [b_*, b^*] &:= [a_* - b_*, a^* - b^*], \quad a_* \leq b_*, a^* \geq b^*. && (4.8)
\end{aligned}
$$

Proof. The proof for Equations (4.6), (4.7) and (4.8) is trivial. □

Propagation of deviation bounds can be implemented according to Lemma 4 using interval arithmetic, compare also Figure 4.9.

Lemma 4. *Let* P *and* C *be graph nodes with* n *children each. Let* $(d_0^P, ..., d_{2n}^P)$ *and* $(d_0^C, ..., d_{2n}^C)$, $d_i^P, d_i^C \in \mathbb{R}$ *be the* $2n + 1$ *duration values of P and C and* $(e_1^C, ..., e_n^C)$, $e_i^C \in \mathbb{IR}$ *the deviation intervals of the sub-trees rooted in C's child nodes. Let the interval* $T \in \mathbb{IR}$ *globally define the maximum absolute deviation of time stamps. Then the maximum intervals* $T_j \in \mathbb{IR}$ *with*

$$
\begin{aligned}
T_j + e_j^C \subseteq &\quad T, \qquad j = 1, ..., n, && (4.9)\\
T_j := &\quad T \ominus e_j^C. && (4.10)
\end{aligned}
$$

---

[2] This is equivalent to managing upper and lower bounds separately.

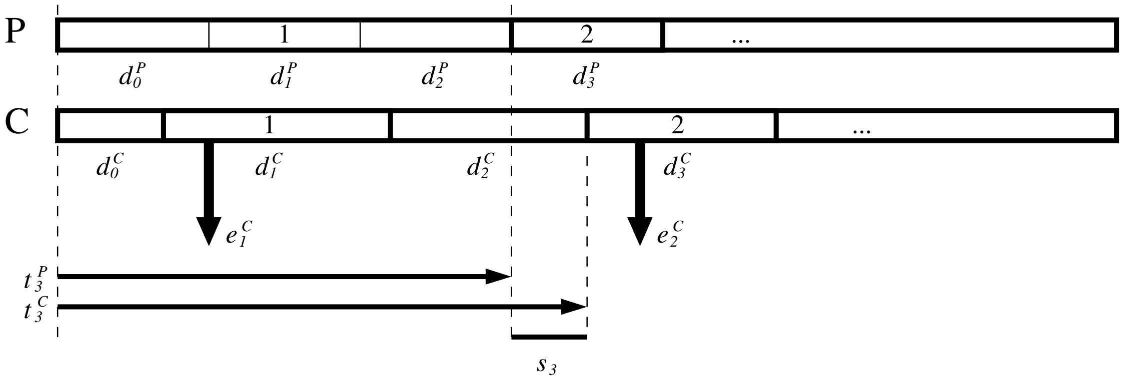


Figure 4.9: Comparison of two nodes P and C for absolute time deviation: For the second child nodes (2), the time deviation is $s_3 = t_3^P - t_3^C$, compare Equation (4.14). This is added to deviation interval e_2^C of the second sub-tree. The result $s_3 + e_2^C$ contributes to the deviation interval E of current node , see Equations (4.16) and (4.17).

are the sufficient local deviation bounds for accumulated deviations in durations. Now, the tests

$$\sum_{i=0}^{j-1} \left(d_i^P - d_i^C\right) \in T, \quad \forall j = 1, ..., 2n+1 \quad \text{and} \tag{4.11}$$

$$\sum_{i=0}^{2j-2} \left(d_i^P - d_i^C\right) \in T_j, \quad \forall j = 1, ..., n \tag{4.12}$$

can be performed with local data only.

Proof. For both nodes P and C the following t_j^P and t_j^C

$$t_j^X = \sum_{i=0}^{j-1} d_i^X, \ j = 0, ..., 2n+1, \ (t_0^X = 0), \ X \in \{P, C\} \tag{4.13}$$

re-create the time stamps of the nodes relative to the node's first time stamp and

$$s_j = t_j^P - t_j^C = \sum_{i=0}^{j-1} d_i^P - \sum_{i=0}^{j-1} d_i^C, \ j = 0, ..., 2n+1, \ (s_0 = 0) \tag{4.14}$$

$$= \sum_{i=0}^{j-1} \left(d_i^P - d_i^C\right) \tag{4.15}$$

are the deviations between the time stamps of P and C. All s_j must be bounded by the global deviation interval T. The deviations in the sub-trees given by the e_j^C are moved to the past or to future by s_{2j-1} as the start time stamp t_{2j-1} of the sub-trees includes alterations. Therefore, the combined deviation must obey the global bound:

$$s_{2j-1} + e_j^C \subseteq T. \tag{4.16}$$

This is assured by $s_{2j-1} \in T_j \subseteq T$ according to (4.9). ☐

If the test for local time stamp deviation from Lemma 4 was successful then node C can be replaced by P. Then the time stamp deviation interval E for node C will be necessary when C's parent node is compared with respect to soft properties.

Lemma 5. *Let C be a graph node that is subject to replacement with properties as defined in Lemma 4. Its deviation interval can be computed from all time stamp deviations in C and in the sub-trees of its child nodes as:*

$$E = \{s_i\}_{j=0}^{2n+1} \cup \bigcup_{j=1}^{n} \left(s_{2j-1} + e_j^C\right). \tag{4.17}$$

For leaf nodes this simplifies to $E = \{0, s_1\}$. If node C is not replaced, E defaults to 0.

Proof. With s_j and e_j^C like in Lemma 4 E is the superset of all time stamp deviations in C. □

The deviation interval E will be part of the comparison of the parent node of C. Always, there is $0 \in E$ since $s_0 \equiv 0$. Note that the e_i^P from replacement nodes are not needed in Lemma 4 or in Lemma 5. For every temporary node C the deviation intervals e_i^C of all child nodes C_i are only required at finalization time of C, i.e. those of recently finalized nodes. It is most suitable to store all child node's deviation intervals in the temporary parent node C instead of the child nodes C_i directly. This means, to every *child node pointer* a deviation interval is supplemented in the temporary node data structure. The permanent notes do not need to contain deviation intervals anymore, thus reducing the total memory consumption. Compare also Figure 4.5 where deviation intervals are associated to the edges of the CCG.

4.2.5 Compression Metrics

The degree of compression achieved can be measured in two ways: Counting the number of final graph nodes or comparing the total memory usage. For both the *compression ratio* R is defined as the uncompressed value divided by the compressed value. The compression ratio according to the graph node count is defined as

$$R_{nodes} := \frac{N}{n} = \frac{\text{nodes in uncompressed CCG}}{\text{nodes in cCCG}}. \tag{4.18}$$

The compression ratio according to memory usage is defined as

$$R_{memory} := \frac{M}{m} = \frac{\text{memory for uncompressed CCG}}{\text{memory for cCCG}}. \tag{4.19}$$

The former reflects the reduction of total effort for graph traversal and evaluation algorithms when constant complexity per node is assumed. The latter reports the raw memory savings.

Both, R_{nodes} and R_{memory} can be determined without establishing an uncompressed version of the CCG. The values of n, N, m and M can be counted in the course of compression. At finalization time of each node C, its memory size m_C is known. Always, N is increased by one and m_C is added to M. If a node is subject to compression, then n and m remain unchanged. If C is not replaced but kept in the CCG then n and m are increased in the same way as N and M. It follows, that compression will never increase node count or memory usage:

$$n \leq N \implies R_{nodes} \geq 1, \tag{4.20}$$

$$m \leq M \implies R_{memory} \geq 1. \tag{4.21}$$

In general, R_{nodes} and R_{memory} are not proportional. They show similar behavior with respect to some parameters, e.g. the time deviation bounds abs and rel, and behave differently with respect to other, e.g. the branching factor b. See Chapter 5 for experimental results.

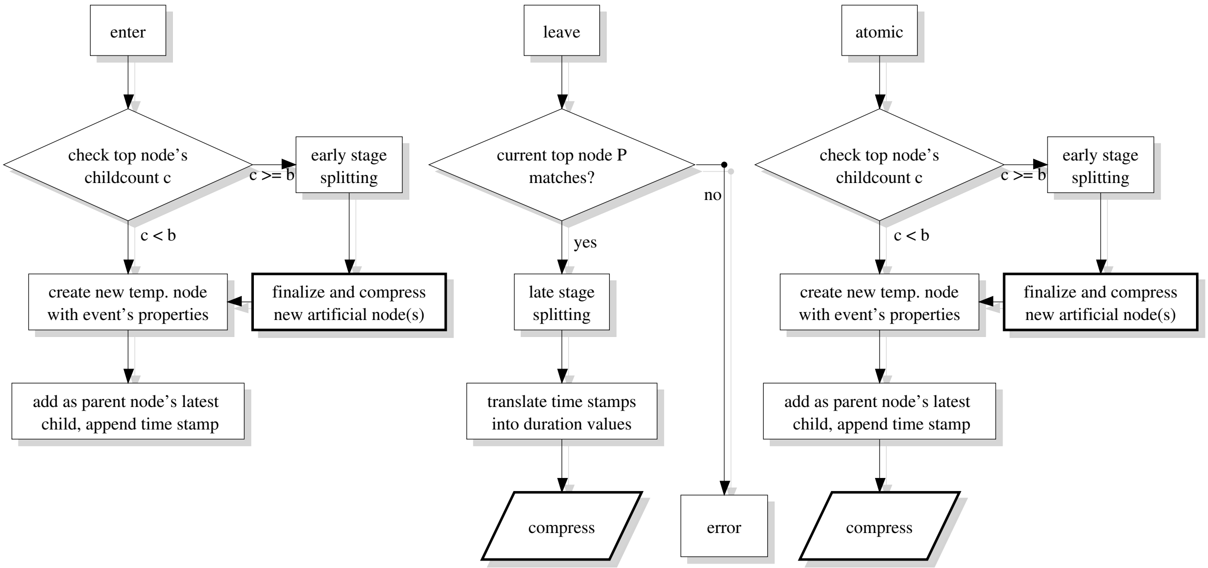


Figure 4.10: Structure of the combined algorithm for CCG construction and compression comprising of three parts for *enter events*, *leave events* and *atomic events*.

4.3 The Combined Construction and Compression Algorithm

For an actual implementation the compression part is embedded in the CCG construction. By this means, at no point a large uncompressed parts need to be stored before compression. All compression steps that affect a particular node are performed at finalization of the very node. This includes the search for an suitable replacement node whereupon either a replacement is found or a new permanent node is created from the temporary node. Figures 4.10 and 4.11 give an overview of the combined construction and compression algorithm. It consists of three parts for *enter events*, *leave events* and *atomic events*. Only the latter two contain actual compression.

For *enter events* an temporary node is created, initialized and appended to the temporary parent node (Figure 4.10 left). An additional check is performed for the parent node's child count in order to trigger a node split operation if necessary (see Section 4.1.2). The creation and finalization of artificial nodes is almost identical to the handling of atomic nodes.

For *leave events* there is a previously created temporary node to be finalized (Figure 4.10 middle). At first, late stage splitting is performed if necessary (see Section 4.1.2). If new artificial nodes are created, they are to be handled separately like explained for atomic nodes below. Then, all time stamps of the current node are translated into time durations b efore the actual compression takes place including the search for replacement nodes (Figure 4.11). If a replacement nodes was found, the temporary node is deleted, achieving actual compression. Otherwise a permanent node is created from the temporary node and added to the set future replacement candidates. Finally, the child reference from the parent node to the current node is updated.

The algorithm for *atomic events* combines parts of both others (Figure 4.10 right). The first part is similar to the one for *enter events*, including the early splitting of the parent node and the creation of a new temporary node. The final part is shared with the algorithm for *leave events* including the actual compression part (Figure 4.11).

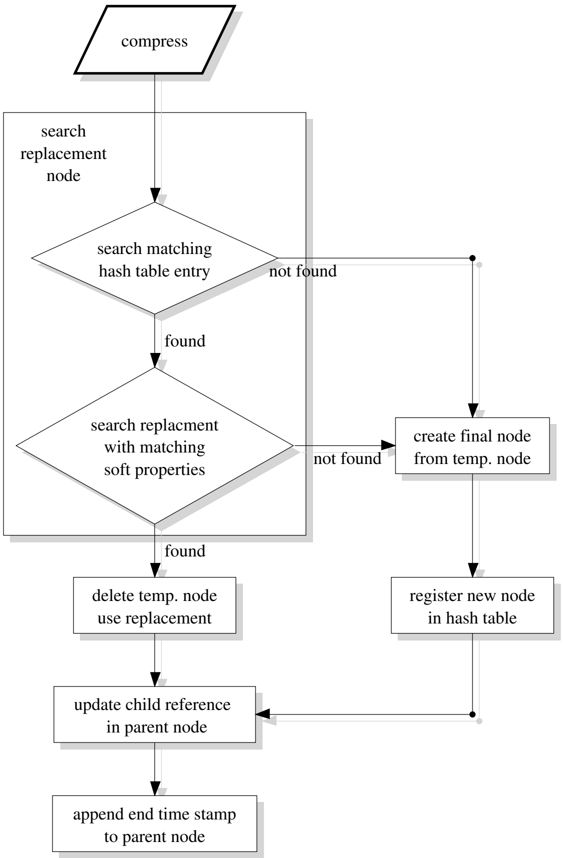


Figure 4.11: The compression algorithm as used by the combined CCG construction and compression.

4.4 Advanced Construction and Compression Techniques

Based on the default construction and compression algorithm discussed above, some advanced techniques can be derived:

Re-compression: Reducing an existing compressed CCG further with enlarged deviation bounds.

Merging: Merging of several (compressed) CCGs or sub-graphs thereof into a joint graph.

Adaptive compression: Determine compression parameters complying given ressource limits.

The following sections present the necessary modifications to the basic CCG construction and compression algorithms.

4.4.1 Re-Compression of Existing CCGs

Starting with a compressed CCG with deviation bounds abs and rel, re-compression requires enlarged deviation bounds[3] $abs_{new} \supseteq abs$ and $rel_{new} \supseteq rel$. If nodes A and B become compatible with respect to abs_{new} and rel_{new} that were not with respect to the original deviation bounds, then the compression ratio can be further increased. It is inevitable to assume that the original deviation bounds have been used to maximum extent, because detailled deviation information is unavailable, compare Section 4.2.4. This is an overestimation for nodes replaced with smaller actual deviation. Therefore, re-compression will achieve inferior compression than original compression with abs_{new} and rel_{new}.

---

[3] Deviation bounds are modeled as intervals instead of scalars, see Section 4.2.4 and Lemma 3.

The re-compression scheme needs to use the following effective deviation bounds for the comparison of nodes A and B with respect to the soft properties:

$$abs_* := abs_{new} \ominus abs, \quad \text{and} \tag{4.22}$$

$$rel_* := rel_{new} \ominus rel. \tag{4.23}$$

The operation $\ominus$ denotes the inverse interval addition, see Lemma 3 in Section 4.2.4.

The CCG re-compression consists of two stages that are to be iterated. The first stage searches for compatible nodes. According to Section 4.2.1, such nodes can only be found in the same S-list. The trivial algorithms would search for pairs of compatible nodes and use either one as a replacement for the other. A more sophisticated scheme would use locally optimal replacement nodes, i.e. nodes that are eligible to replace the largest number of other nodes. Both schemes would cause $O(l^2)$ effort with the length of the S-list l. An even more ambitious approach selecting globally optimal replacements, which make parent nodes (and parent's parents, etc.) compress optimally, would be exceedingly expensive in terms of computational complexity.

During the first stage, all replacements of a node B by node A are recorded in a translation table $T : B \rightarrow A$ which requires further temporary memory of order $O(n - n_{new})$ with the current node count n and the subsequent node count n_{new}.

The second stage of node re-compression performes the translation of child node referencing according to the table T. It adjusts all child references to any of the previously deleted nodes B in all nodes. If at least one child reference was changed inside a node, an additional re-hashing must be performed, because child references are involved in the computation of hash value. After the second stage the translation table T is reset. The effort for checking all child references is at least $O(n \cdot b)$ because for every node there are at most b children. The table look-up in T as well as the re-hash operation are assumed to have constant effort $O(1)$. Both stages of the CCG re-compression algorithm are shown in Figure 4.12.

If a node is moved to a new S-list, there is potential for further compression within this S-list. Therefore, the the two stages need to be iterated until no new replacements occur in order to produce a completely re-compressed CCG. The iteration is guaranteed to terminate according to the following lemma.

Lemma 6. *Re-compression of Compressed Complete Call Graphs terminates after d iterations with the maximum depth of the graph d.*

Proof. A node p is subject to re-compression after the latest child B has been re-compressed whereupon the respective child reference is translated. Assume B was re-compressed in iteration i, then P appears in the right S-list in iteration $i + 1$ and can be replaced. The index i is the order of a node.

Leaf nodes have order 1 and can be replaced in the first round. All nodes of order 2 are only dependent on leaf nodes. Thus they can be replaced in the second round and so on. A maximum oder node, which has no parent node by definition, is subject to re-compression in round d. Deviation intervals are propagated from leaf nodes upwards in the very same scheme. ☐

In order to guarantee absolute global deviation bounds, the deviation intervals of all replaced nodes need to be tracked, compare Sections 3.3.2 and 4.2.1 as well as Lemma 5. This requires $O(n)$ temporary memory in a separated data structure, because permanent nodes contain no deviation information like construction nodes.

In Section 4.1.4 contiguous memory placement for node objects was introduced which does not allow to deallocate memory selectively. Therefore, re-compression could not reduce the memory consumption m but would cause memory fragmentation. The node count n is reduced nonetheless. This could be solved by new allocation and copying requiring additional temporary memory as well as another iteration of node re-referencing stage.

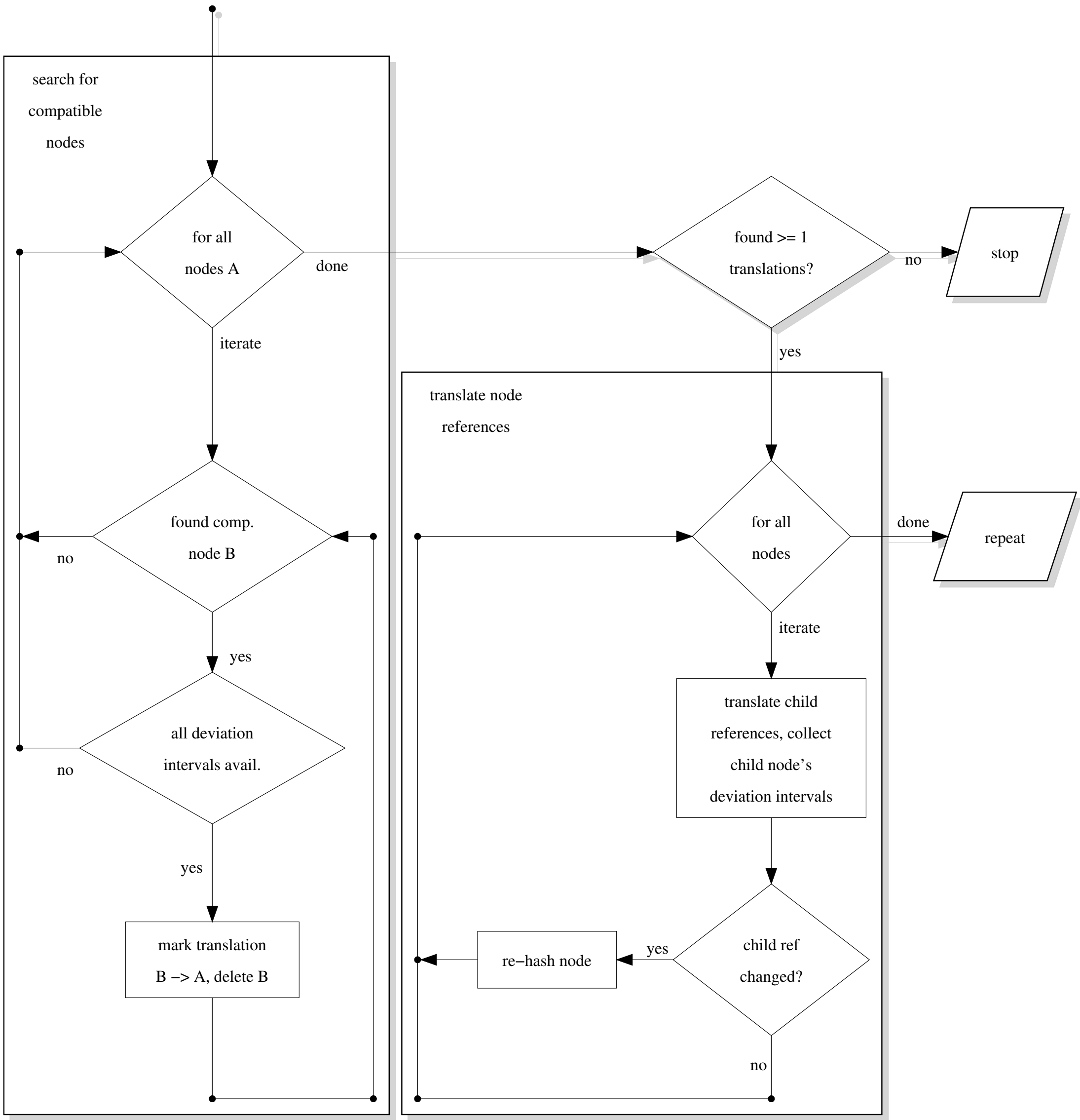


Figure 4.12: The CCG re-compression algorithm including translation of node references. It is repeated as long as new translations are found.

The total effort of the re-compression algorithm comprises from the two stages that are iterated $\leq d$ times. The first stage compares every node with all $l \leq L$ nodes in the same S-list with $O(n \cdot l)$ effort. The second stage needs $O(n \cdot b)$ effort as stated above. This results in a total effort of

$$O\left(d \cdot n \cdot (l + b)\right). \tag{4.24}$$

It relates to $O(N \cdot l \cdot b)$ effort for CCG construction and compression according to Section 3.3.3. This indicates no notable advantage for re-compression of an existing CCG over compressing the original data with abs_{new} and rel_{new} in the first place. Instead, the disadvantages indicated above makes re-compression generaly inadvisable, compare also Section 5.3.1.

4.4.2 Merging of Disjoint CCGs

Assuming there are two or more compressed or uncompressed CCGs without shared sub-graphs (disjoint CCGs) and there are unique child identifiers (pointers) between all G_i. Strictly speaking, the $\{G_i\}_i$ form a common CCG G_* already, yet with unconnected components. True merging should create connections between the original graphs G_i such that sub-trees become shared between previously disjoint sub-graphs reducing the total memory consumption. However, the existence of identical sub-trees in multiple G_i is rather improbable, in particular for large sub-trees which are more promising in terms of compression. Compare the arguments for a single node in Section 3.3.2 and Equation 3.3.2.

As a consequence, it is suggested to combine merging with re-compression. This implies the same disadvantages in terms of overhead and reduced compression as for re-compression, see Section 4.4.1.

4.4.3 Adaptive Deviation Bounds

So far, deviation parameters have been specified explicitly in order to have node count and memory usage reduced by an amount which is not precisely predictable. For some applications it is desirable to explicitly limit resource usage (memory consumption) while the exact deviation bounds are to be defined implicitly. Through this means, it is possible to analyze a given trace with the (approximately) minimum deviation bounds that comply with the available resources.

The general algorithm for adaptive compression keeps track of the resource usage while following the conventional construction and compression procedure. When the given resource limit is exceeded, the algorithm concludes that the current deviation bounds are too restrictive.

Then, the adaptive algorithm needs to perform two tasks in order to resume construction and compression: expanding the deviation bounds and reducing the current resource usage. Both are discussed below. If certain maximum deviation bounds are not sufficient to compress the given trace in accordance with the resource limits, the algorithm must abort.

Expansion Strategies

Adaptive compression always starts with conservative initial deviation bounds or zero deviation. The expansion of current deviation bounds to more relaxed bounds can be done according to several strategies:

- explicitly scheduled,
- arithmetic expansion,
- geometric expansion, or
- dynamic expansion.

For the first method an explicit list of deviation bounds p_i must be specified, which is monotone increasing $p_i \subsetneq p_{i+1}$ for generic deviation parameters $p_i \in \mathbb{R}$. This approach might be usefull for special purposes, yet in general, it is rather inconvenient.

The second method expands the deviation bounds by a constant offset e per iteration. This results in a linear expansion strategy $p_i = p_0 + e * i$. The successive growth from small scale compression bounds to medium and large scale compression parameters will require many iterations for small e, compare Sections 5.2.1 and 5.2.2.

The third method uses a constant expansion factor f for the current deviation bounds. It results in an exponential growth $p_i = p_0 * f^i$. Compared to the second strategy, this allows a slow increase for small scale compression parameters and an accelerated growth for large scale parameters.

Unlike the three former static methods, a dynamic expansion strategy can anticipate the final ressource usage U_{total} by:

$$U_{total}^{p_i} = \frac{d}{c} \cdot U_{current}^{p_i}, \tag{4.25}$$

if progress information is available during reading: For example, if the numbers of previously processed events c and total events d are given. The term $U_{current}^{p_i}$ denotes the current resource usage. With an appropriate compression model, the parameter p_{i+1} can be predicted such that $U_{total}^{p_{i+1}} \approx U_{avail}$, compare also the compression model in Section 5.1.1.

Even though, this estimation may be very coarse, it allows to perform bigger expansion steps in the early phase of a trace and smaller steps near the end.

Resumption Strategies

After updating the deviation bounds, there is the task of reducing the current resource usage in order to proceed in compliance with the ressource limits. Two resumption strategies are proposed, compare also Figure 4.13

- adaptive re-start or
- adaptive re-compression

The adaptive re-start strategy simply dismisses all current data and restarts construction and compression from the beginning of the trace. This eliminates the current ressource usage completely but abandons the previous compression effort.

The alternative re-compression strategy employs the CCG re-compression algorithm as introduced in Section 4.4.1. By this means, the previous effort is not wasted but re-used. However, repeated re-compression would multiply the negative effects of re-compression. Furthermore, additional temporary memory would be required in a situation where memory consumption is critical.

Optimal Deviation Bounds

The previous algorithm for adaptive compression find sufficient deviation bounds to compress a given trace with limited ressources, provided such parameters exist. The following extension allows to find *nearly optimal* deviation bounds, this means the most restrictive bounds that comply with the ressource limits except for accuracy $\varepsilon > 0$. For this purpose an expanding strategy is not suitable, because it would be restricted to unreasonably small expansion steps of ε. Instead, an interval inclusion for the optimum deviation parameters is proposed.

It requires two arbitrary parameter sets p_{inf} and p_{sup}, such that compression with p_{inf} will exceed the allowed ressources[4] and compression with p_{sup} complies with the ressource limits.

Then, the optimal parameters can be determined by successive interval bisection as follows: Determine a *midpoint* parameter setting p_{mid} with $p_{inf} < p_{mid} < p_{sup}$. If compression succeeds before ressources are exceeded, then the next iteration is started with $p_{inf} := p_{mid}$, otherwise, compression is aborted and the iteration continues with $p_{sup} := p_{mid}$. The iteration stops if $|p_{sup} - p_{inf}| < \varepsilon$ is reached.

In case of parameter intervals, the '$<$' relation needs to be replaced by the inclusion relation '$\subset$'. Then the iteration uses parameter intervals $p_{inf} \subset p_{mid} \subset p_{sup}$. If multi-dimensional vectors of parameters are used instead of scalars, then the '$<$' relation needs to be applied per component. The multi-dimensional midpoint can be computed over all dimensions simultaneously or in a single dimension at a time in an alternating manner. See [AH83] for both cases as well as for their combination, in particular the Chapter *Methods for the Simultaneous Inclusion of Complex Zeros of Polynomials*.

---

[4] It is not necessary to actually complete compression with this parameters. Detecting the fact is sufficient.

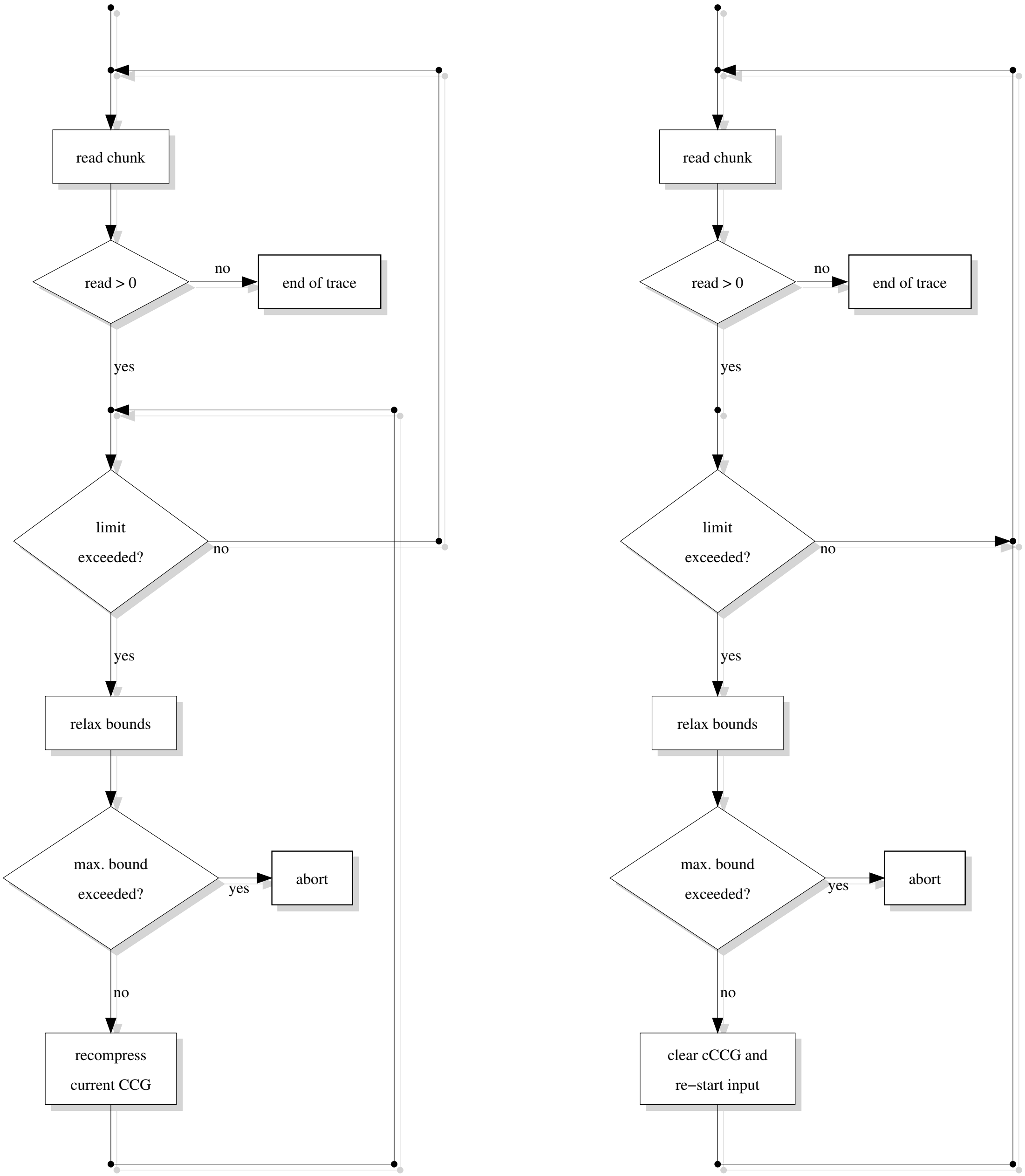


Figure 4.13: Algorithms for compression with adaptive deviation bounds: The re-compression approach (left) produces high overhead and has only reduced compression ability. The re-start approach (right) allows maximum compression with less overhead. Note the similar components but the different repetition structure.

4.5 CCG Analysis Algorithms

After construction and compression of CCGs, evaluation and analysis algorithms for the CCG data structure have to be designed. This section covers the most important evaluation operations which may be used either as building blocks or as model for further evaluation methods. There are two basic categories. The first category handles single *positions* in a CCG, that means positions at graph nodes or positions relative to the original sequence of trace events:

- random access to positions in the CCG,
- linear traversal over positions, and
- timestamp search.

The second category contains evaluation algorithms for ranges of positions:

- statistical summaries,
- timeline rendering, and
- send-receive matching.

The latter kinds of evaluation operations use position information as input and provide specific result information as output.

4.5.1 The Random-Access Iterator

Random access iterators are objects with the ability to conveniently navigate in a container data structure, hiding away implementation details. It can be regarded as a *smart pointer* that is able to proceed from the current element to a successor or a predecessor element. Furthermore, an iterator is not supposed to alter the referred data structure in any way, even though, the data structure may be changed by operations that are using the iterator.

Inside a CCG there are two directions for iterating, i.e. two separate successor/predecessor relations. One one hand, regarding graph nodes, which have an inherent parent/child direction. This uses the semantics of a tree graph. On the other hand, regarding the original events which have a natural before/after direction. While events in CCGs are represented by positions inside a graph node, the event-related iterator emulates a linear list of events.

Traversal in both directions can be implemented by the same iterator class. The directions are named like following, compare also Figure 4.14 for an illustration:

- *up*: proceed to the parent node
- *down(i)*: proceed to the $i'th$ child node
- *forward*: proceed to the next event
- *backward*: proceed to the previous event

Mind, that for compressed CCGs there is no *unique parent node* property anymore, thus, the *upward* direction is undefined. Instead, it is considered as returning in the same way the iterator traversed its path downwards before. Therefore, an iterator object is made up of a stack containing CCG nodes as they have been traversed on the way from a root node to the current node. The four operations *up*, *down*, *forward* and *backward* are implemented as follows:

up: Remove the top-of-stack element, i.e. return to the parent of the current node. This is allowed only, if the current node is a non-root node. After an *up* operation the iterator position points right after the reference to the previous node.

down(i): Push the i'th child of the current node to top-of-stack making it the new current node. This is only valid if there is an i'th child to the current node. The iterator position will point to the beginning of the new top node.

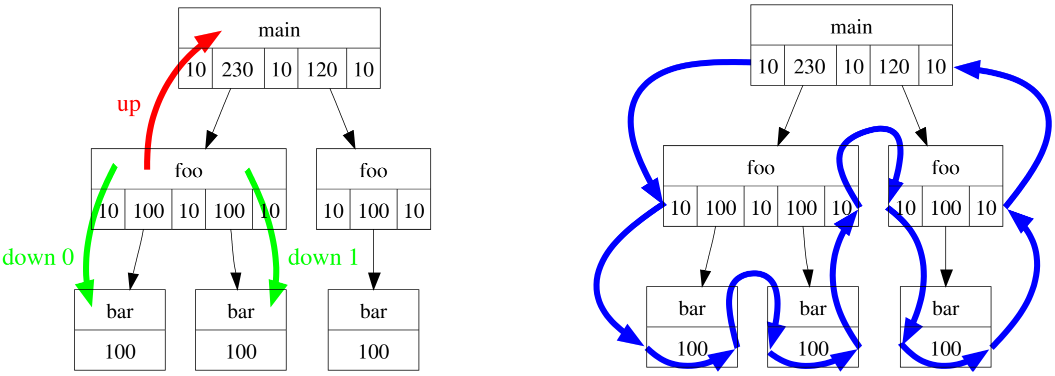


Figure 4.14: The left hand side shows how an iterator moving upwards (red) or downwards (green) on a CCG (node semantics). The right hand side shows the complete path of the forward traversal (blue) through the events of a CCG (event semantics).

forward: Proceed to the next following event, handling different cases:

- If the iterator points to the beginning of a node with children the iterator will proceed to the beginning of the first child, like the *down(0)* operator would.
- If the iterator points to a node's beginning and there are no children then the iterator proceeds to the node's end.
- If pointing to the end of a node, an iterator would proceed to the begin of the next child node of the same parent node.
- If no more child nodes are left in the current parent node, the iterator proceeds to the parent's end position.

backward *Backward* traversal is the inverse of *forward*.

The random access iterator is also capable of tracking time stamps of events. Assumed, the start time of the current node is known, then the start time of every child node can be computed from local information by adding all time duration values before the the particular child, compare Figure 3.5.

The four iterator operations presented here, are used as basic elements when constructing more complex evaluation algorithms. The computational effort of *up*, *forward* and *backward*, is assumed to be constant $O(1)$ while the operation *down(i)* has $O(i) \leq O(b)$ complexity, compare Section 3.2.2.

4.5.2 Timestamp Search

The objective of *time stamp search* is to find the very graph node containing a certain time stamp. Within every process graph this has got an unique solution provided the time stamp lies inside the total scope of this process. The result is an iterator object pointing to the node of interest.

Figure 4.15 shows the recursive algorithm which starts with the root node. The time interval of the current node contains the target time-stamp t. If exactly one ot the time intervals of the direct child nodes covers t as well, then the algorithm continues with this node. Otherwise, the current node is returned as result.

The computational effort of this search operation is determined by the number of recursive steps which is bounded by the maximum tree depth d. Every recursion step involves the traversal of the children of the current node, resulting in $O(d \cdot b)$ complexity.

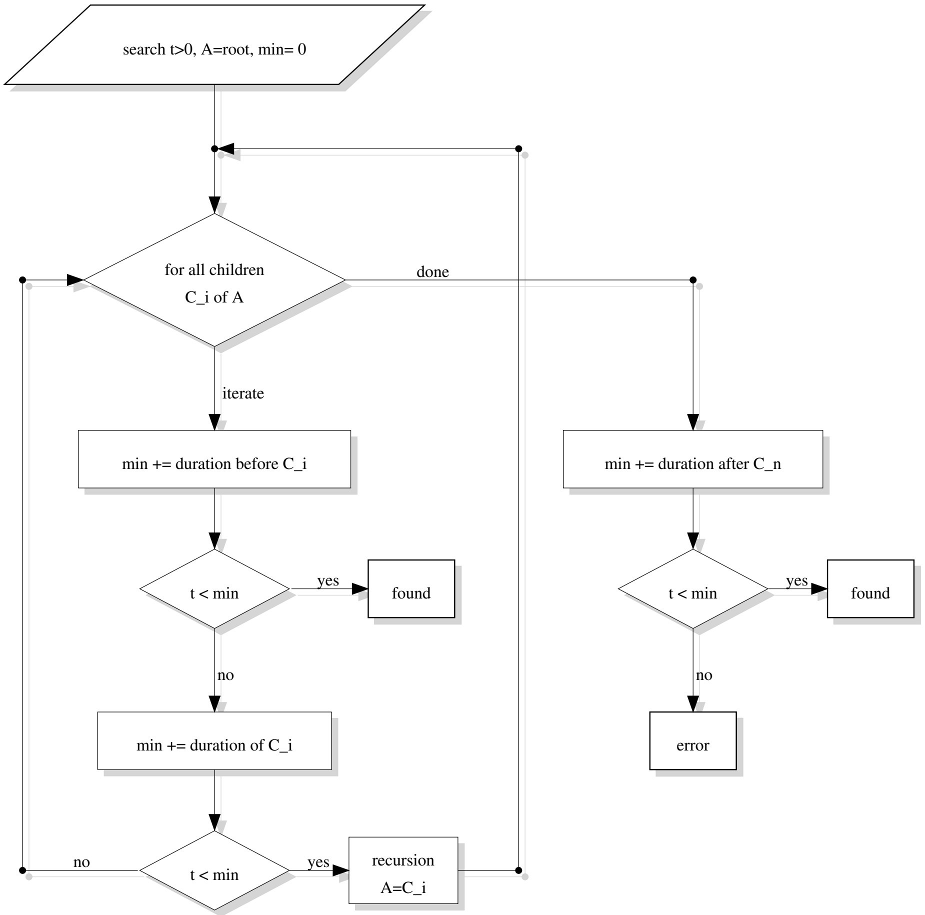


Figure 4.15: Recursive algorithm for fast time-stamp search in a CCG.

4.5.3 Summary Query Algorithms

One of the most important and most frequently used evaluation operation is the so called *summary query*. It provides a summary over certain properties of events in a given time interval and a set of processes. Examples for properties covered by summary queries are exclusive or inclusive run time per function, number of calls or sub-calls per function, message count or volume, I/O volumes and many more.

In general, summary queries are applicable to all properties which provide additivity, i.e. the sum of partial results for disjoint subsets equals the result for the superset:

$$P(A) = \sum P(B_i) \qquad \forall i \neq j : B_i \cap B_j = \emptyset, A = \bigcup B_i. \tag{4.26}$$

On one hand, summary queries are utilized to provide a coarse overview of a trace. This can cover questions like *Which functions cause the major computational load?* or *Which processes show above-average communication?*

On the other hand, a more fine grained overview of the temporal and spacial distribution of certain properties can be obtained with multiple summary queries for adjacent time intervals. Figure 4.16 shows an example of a color coded timeline diagram displaying the number of state changes per time segment.

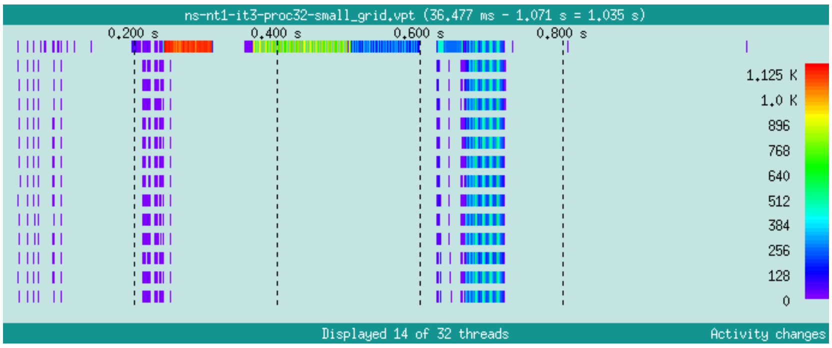


Figure 4.16: Vampir Server's Color Coded Timeline diagram displaying frequency of certain trace events over time. This example reveals high rates of state changes in the first process.

Caching Queries on Uncompressed CCGs

At first, the recursive evaluation algorithm is applied to uncompressed CCGs. In order to compute the partial result for any graph node A, the respective partial results of all child nodes are summarized and maybe supplemented by additional information from A itself.

This fits most naturally to the structure of the CCGs. Some additional precautions are necessary for nodes that intersect with the boundaries of the query time interval. Figure 4.17 shows the structure of an uncompressed CCG of a single process, which at the same time resembles the evaluation graph of the recursive query.

When extending the scope from single queries to successive queries, the recursive approach can be enhanced by a caching scheme. Assuming there are overlapping time intervals within successive queries then there are graph nodes evaluated repeatedly for the same partial results.

Redundant computation of partial results can be avoided by introducing caching. Once a partial result has been computed, it is inserted into the cache in order to avoid future re-computation. Only nodes intersecting with the bounds of the query time interval must be treated separately and are excluded from the caching scheme. Thus, all nodes intersecting with the interval bounds must be evaluated in the conventional way. This causes only $O(d)$ effort with the tree depth d, because there are at most d nodes intersecting with each of the two interval bounds.

The memory consumption for complete caching is $O(N)$. In order to limit the cache size one might select only a sub-set of the results to be cached. There are several heuristics available:

1. Select every c'th item in order of arrival, covering $1/c$ of all graph nodes on average.
2. Select all nodes with depth levels $l(\mod c) = 0$, covering $1/c$ of all graph nodes on average.
3. Select all nodes which are more often referenced than a certain threshold a, which can be combined with the former heuristics. There is no fixed share of cached nodes.
4. Fixed size caches with replacement strategies, like LRU (Least Recently Used) or LFU (Least Frequently Used).

Assuming the cache strategy (2) is used, the cache memory requirements are like $O(N/c)$. The evaluation effort is reduced from $O(N)$ for the initial query to $O(d + b^c)$ for successive queries where d is the tree depth and b the branching factor. The term d derives from the fact, that few nodes intersect with the two interval borders. Those nodes are connected by two *critical paths* with not more than d intersecting nodes each, which are excluded from caching, compare Figure 4.18. The term b^c expresses the need to traverse up to c depth levels before a cache hit terminates the recursion.

The same average complexity applies to caching strategy (1) but with a unbounded worst case scenario. Both, (1) and (2) will profit when combined with (3), which would slightly increase the memory requirements but reduce the average computational effort notably depending on the actual threshold a.

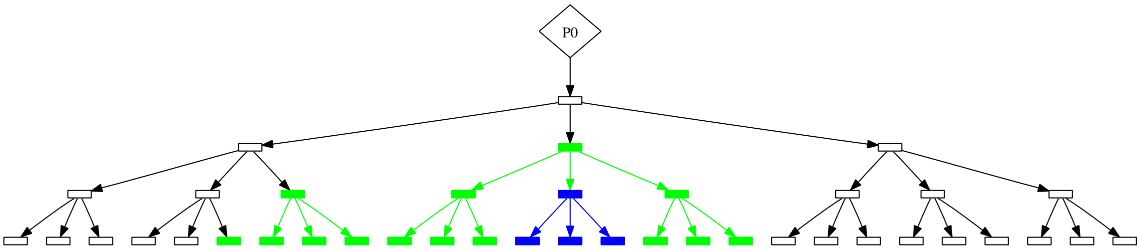


Figure 4.17: Uncompressed CCG with multiple evaluation paths. Initial evaluation involves all graph nodes. Zooming in to a time interval requires re-evaluation of some nodes (colored nodes). With caching nodes' partial results can be re-used, terminating recursive re-evaluation. The same applies for the next zoom level (blue nodes).

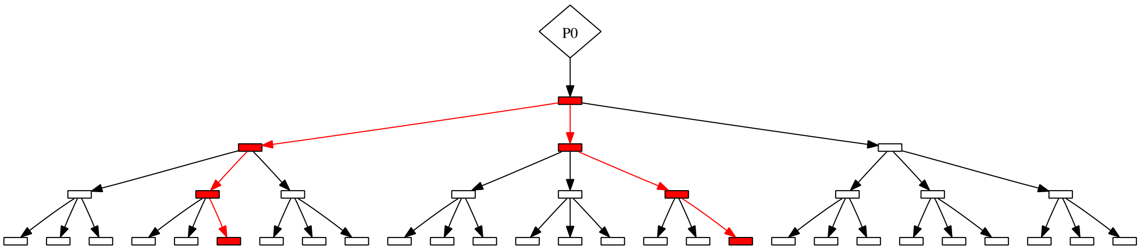


Figure 4.18: Uncompressed CCG with *critical paths* along the time interval borders marked red, corresponding to the second query in Figure 4.17.

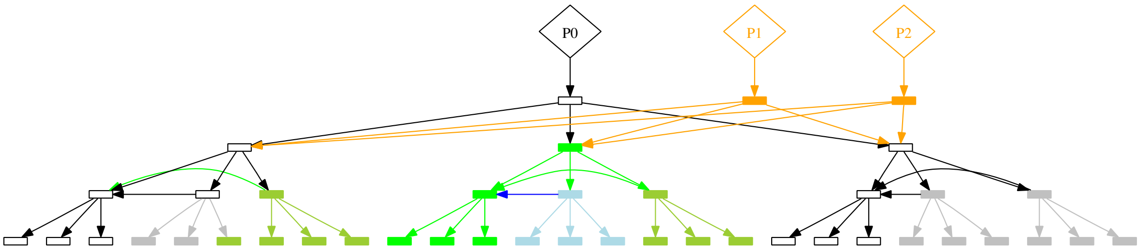


Figure 4.19: Compressed CCG with multiple evaluation paths corresponding to the uncompressed example in 4.17. Some sub-trees are replaced by references to other sub-trees. This allows to re-use partial results from cache even during an initial query. Furthermore, sub-trees can be shared among CCGs of multiple processes (orange).

Caching Queries on Compressed CCGs

The same caching scheme as described before can also be applied to compressed CCGs The computational effort for initial queries is reduced from $O(N)$ without caching (for uncompressed as well as compressed CCGs) to $O(n)$ with complete caching or to $O(n + b^c)$ with partial caching like before. For successive queries the first term vanishes. Like before the depth d characterizes the effort to traverse all non-cache-able nodes along the time interval boundaries. Thus, the computational effort is $O(d + b^c)$. The cache memory requirement is now $O(n/c)$ instead of $O(N/c)$. This reduction equals the node compression ratio R_{nodes}, compare Section 4.2.5.

Thus, the cache efficiency is increased when used with compressed CCGs instead of uncompressed ones, because fewer cache entries are re-used more frequently. As an example, Figure 4.19 shows successive queries to an compressed CCG which is derived from the uncompressed graph in Figure 4.17. This example also shows how the caching mechanism can be extended over multiple CCGs of parallel processes.

4.5.4 The Timeline-Rendering Algorithm

The timeline rendering query generates a visual representation of function activity inside a process over time. Functions are associated to groups, that are displayed with the same color. Rectangles of appropriate color mark the activities of functions over a period of time. Those are either arranged as a colored bar per process (compare Figure 3.7) or reproduce the call stack in the vertical direction (compare Figure 3.8). The rest of this section focuses on the *call timeline* while the *process timeline* is regarded as a special case ignoring the call depth.

As indicated in Section 3.4.4, the call timeline has to deliver a visualization for a given time interval $T = [t_{min}, t_{max}]$ with a horizontal pixel resolution of w and t_{min} and t_{max} aligned to the pixel raster. For motivation assume that there are much more events in T than pixels. This problem is mapped to the *cached summary query* by performing column-wise queries for sub-intervals of time $S_i \subset T$ that are associated to the pixel raster, compare Figure 4.20:

$$S_i := t_{min} + [i \cdot h, (i+1) \cdot h), \qquad \forall 0 \le i < w, \quad h = \frac{t_{max} - t_{min}}{w}. \tag{4.27}$$

The subject of the *cached summary queries* on S_i is a mapping of function groups G and call levels j to exclusive run-time $e_i : (G, j) \to \mathbb{R}$. Let F be a function call denoted by enter and leave events, then

$$e_i(G, j) = \sum_{F \in G \wedge level(F) = j} |F \cap S_i| . \tag{4.28}$$

This can be computed efficiently by a summary query, compare Section 4.5.3. For CCGs without deviations in run-time information it is assured that

$$s_i = \sum_{\forall j, \forall G} e_i(j, g) \le |S_i| . \tag{4.29}$$

In consideration of time stamp deviations this needs to be altered to:

$$s_i = \sum_{\forall j, \forall g} e(j, g) \le (1 + R_{upper}) \cdot |S_i| . \tag{4.30}$$

Once all $e_i(G, j)$ are known, the pixel colors can be computed according to a linear color blending as the inner product

$$C_{i,j} = C_{def} \cdot \frac{E_{i,j}}{|E_{i,j}|}, \qquad E_{i,j} := \begin{pmatrix} e_i(1,j) \\ e_i(2,j) \\ \vdots \end{pmatrix}, \quad C_{def} := \begin{pmatrix} (r_1, g_1, b_1) \\ (r_2, g_2, b_2) \\ \vdots \end{pmatrix} \tag{4.31}$$

where C_{def} is the color definition vector for the function groups. Its entries may be composed of color components like RGB values. Normalization is necessary, because Equation (4.30) is not an identity. Alternative, non-linear color blending models are applicable as well.

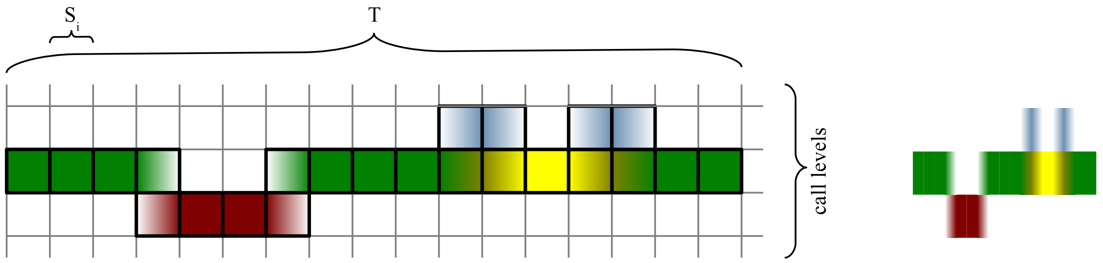


Figure 4.20: Timeline rendering to a pixel raster according to statistics of exclusive run-time per function group. Multiple activities within a single pixels range are visualized by blending the respective colors. Left hand side shows a magnification of the version on the right.

Computational Effort

The over-all effort for computing timeline visualization data is composed from w single cached summary queries. The complexity for a single cached summary query is $O(n_i)$ for initial queries and $O(d + b^c)$ for successive queries, compare Section 4.5.3, where n_i is the node count for the time interval S_i only. Furthermore, with compressed CCGs the query for sub-interval S_i might rely on cached intermediate results from all previous sub-intervals $\{S_j\}_{j<i}$.

The total effort for the computation of the timeline visualization is $O(n_T)$ in worst case with the node count n_T for time interval T. Taking advantage of caching, the effort is reduced to

$$O(w \cdot d + w \cdot b^c). \tag{4.32}$$

Note that this is not explicitly dependent on node count n_T, but implicitly via d. The actual graphical rendering of the results is excluded here.

4.5.5 The MPI Send-Receive Matching Algorithm

The algorithm for matching send and receive events as introduced in Section 3.4.5 needs to iterate through all send and receive events in all processes. Within each process, the temporal order of the events has to be maintained but over parallel processes the order is free. For every current message event C, either the peer event P (in a different process) has been encountered before or will be later. A list of pending messages will store the earlier one until the arrival of the later. In the former case, P will be found in the list of pending messages of the other process. Then it is removed from the list and the pair (C, P) is reported as part of the result. In the latter case, C is appended to the local list of pending events. It will be searched as the pending event as soon as the peer event is handled.

Figure 4.21 illustrates the algorithm. For simplification it is assumed that all event types besides send and receive are ignored. The data structure for temporarily storing pending events maintains a separate FIFO lists for every value of the 5-tuple of properties:

$$(\mathrm{sender}, \mathrm{receiver}, \mathrm{communicator}, \mathrm{tag}, \mathrm{issuer.}). \tag{4.33}$$

This relates to the properties relevant for MPI message matching, see Section 3.4.5. By this means, linear search for replacement nodes is eliminated completely. As soon as the correct FIFO list has been identified, the first entry will be the matching peer event.

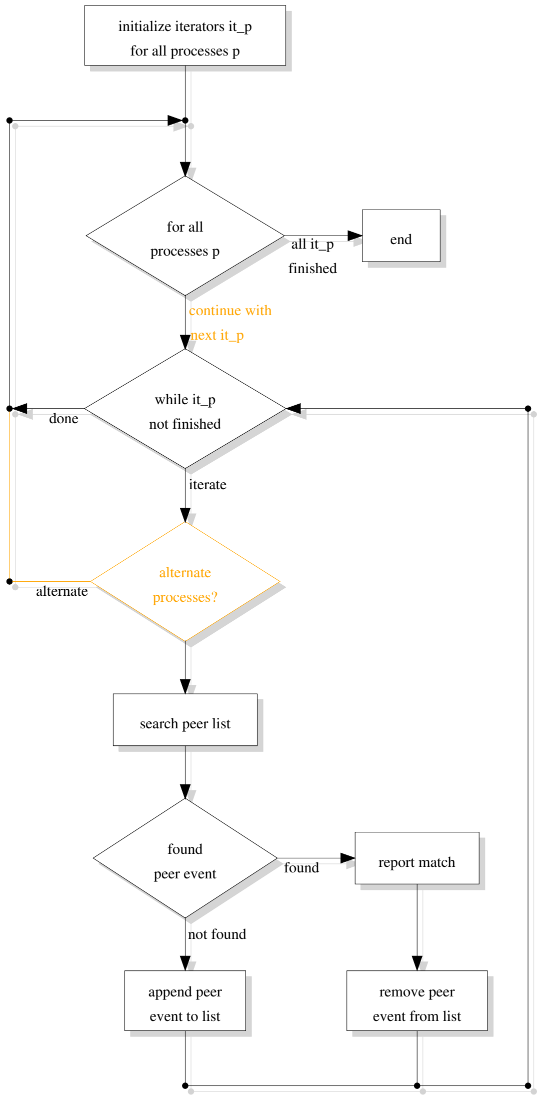


Figure 4.21: Algorithm for matching of send and receive events. The highlighted extension (orange) allows for an alternating iteration over parallel processes.

The algorithm may consider all parallel processes or only a sub-set. In the latter case, all events referring to excluded processes need to be ignored. The order of traversing process traces is insignificant. Yet, it will influence the order of the resulting event pairs. With respect to performance and temporal memory consumption, an alternating scheme might be preferred. A minor modification which highlighted in Figure 4.21 can achieve this. Instead of traversing one process after another, it is done in an interleaving manner which allows notable decrease in temporary resource consumption, compare Section 5.5.

4.6 Persistent Storage and Restoring

Section 3.5 introduced the general idea for persistent storage and restore operations. Below, the detailed algorithms for both operations are discussed. In particular, the order of storing the graph nodes is designed to allow an efficient restore operation.

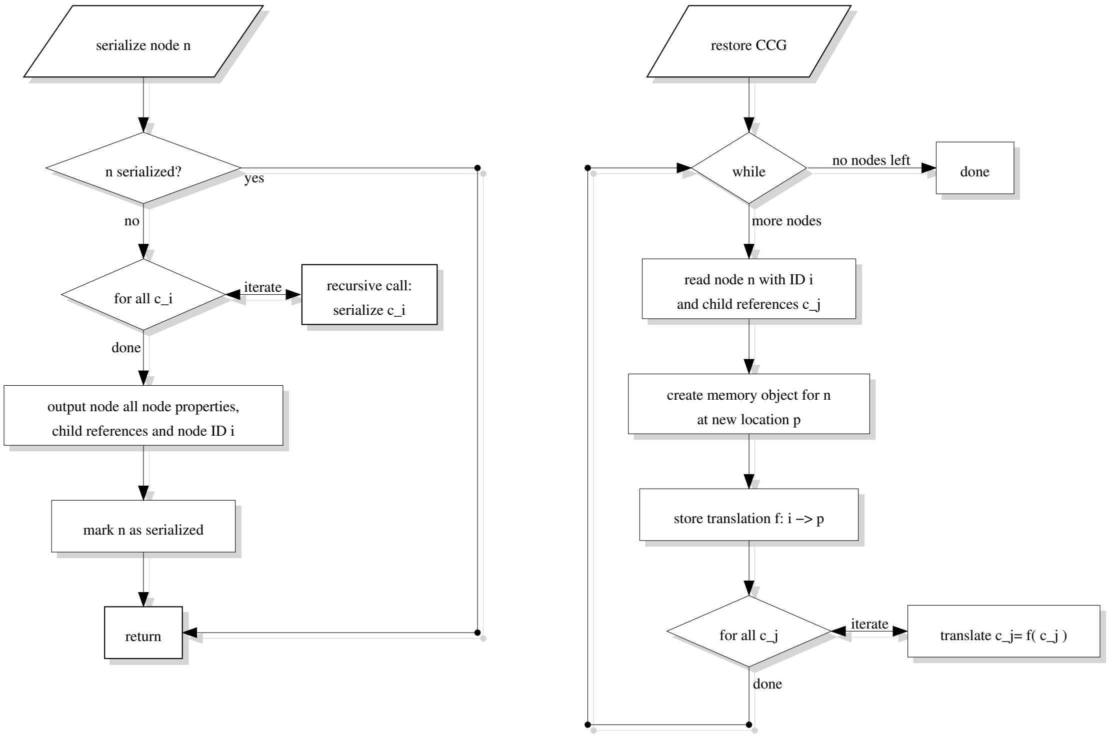


Figure 4.22: Algorithms for the serialization (left) and restore operations (right) for CCGs.

Serialization

Storing a data structure into a file is also referred to as *serialization* [Eck95], because it transforms an arbitrary data structure to a serial representation which is not necessarily present in the first place. For the present application, the order of serializing graph nodes is chosen according to the requirements of the restore operation, see Section 4.6. The particular order is bottom-up such that child nodes are processed before their parent nodes.

Cross references or pointers inside the data structure require special consideration. Their values (addresses) are no longer valid in the serialized stream nor in a restored instance. Therefore, any pointers need to be transformed to identifiers, and all objects need to be supplemented by their respective identifiers such that the references can be re-establish during restore.

The resulting serialization algorithm for CCGs works recursively like shown in Figure 4.22(left). Initially, the serialization operation is started for every root node of a parallel CCG. In the course of recursive traversal of the CCG, every node has to be written to file exactly once, even if it is referenced multiple times. Therefore, it has to be marked after the first encounter.

In case the existing CCG may be destroyed during serialization, then nodes can be marked by overwriting with invalid contents. Otherwise, a hash table (with $O(1)$ access) or a search data structure (with $O(\log n)$ access) has to be maintained holding all nodes already accomplished. Besides the additional effort, the extra memory requirements for this bookkeeping makes the destructive approach favorable.

Writing of the single nodes is done in a straight forward way. All members of the node data structure are written consecutively as a line of an ASCII text file preceded by its identifier and a node type specification. This format is chosen because it is quite robust and avoids platform dependency issues altogether. Standard lossless ZLib compression [lGA02] is applied to the output for convenience.

The transformation of child pointers to identifiers is achieved by type-casting the pointer addresses to an integer of equal size which is guaranteed to be unique within the same memory address space. For a distributed scheme, an arbitrary bijective mapping is necessary to map pairs of process identifiers p and pointers m to unique identifiers i:

$$(p, m) \in [0, p_{max}] \times [0, m_{max}] \longrightarrow i \in [0, n] \tag{4.34}$$

Writing a single node of fixed size to file is assumed to take constant time. Therefore, the over-all effort for serialization of a CCG with n nodes is linear $O(n)$ or almost linear $O(n \log n)$.

Restore

The CCG restore operation involves two steps for every node: recovering of the single nodes and re-establishing of cross-references to the child nodes, see also Figure 4.22(right).

Processing the single nodes works straight forward again: from every serialized node create a memory object of the correct type and fill in all members. Optimizations like placement in a pre-allocated memory areas (Section 4.1.4) or sophisticated encoding via template meta programming (Section 4.1.3) are still feasible. In addition, the mapping $T : i \rightarrow p$ from the node identifier i to the new memory address p needs to be recorded for the second step. After node re-creation, all child node identifiers i are replaced by their corresponding pointers p according to the mapping T. It is guaranteed that the mapping for all child nodes is present by this time, because of the determined serialization order, see Section 4.6.

The mapping table T of size $O(n)$ needs to be stored explicitly during the re-store phase and can be discarded later. However, the restroe operation does not require the hash table structure for look-up of replacement nodes, compare Section 4.2.1. The computational effort for restoring a CCG with n nodes is $O(n)$ if T is maintained as an hash table with $O(1)$ access.

File Compression

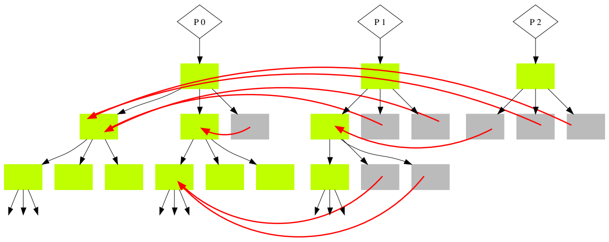
With the serialization and restore operations, the in-memory compression scheme can be extend to a trace file compression method. The compression step would consist of reading the classic trace file, constructing and compressing the CCG in memory, and saving the data structure to a compressed output file. The decompression step would re-create the CCG from the compressed file, traverse all events in temporal order and re-construct a classic trace file. All effects of lossy and lossless compression in memory apply to file compression in the same way.

4.7 CCGs with Distributed Data

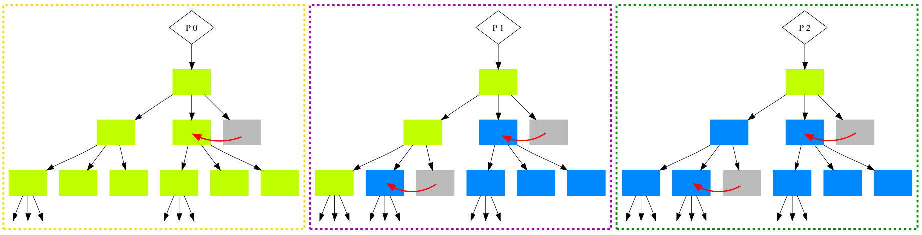
Performance trace analysis is a very data-intensive matter and the CCG data structure is dedicated to relieve the enormous main memory requirements. Nevertheless, the memory consumption might still exceed the available resources, particularly with regard to low or zero deviation bounds and moderate compression ratios.

This section introduces a distributed storage approach using CCG data structure. This concerns distributed construction and compression as well as distributed evaluation.

With traditional data structures, trace analysis is usually memory-bound, this means memory is the limiting ressource while computational effort is of minor importance. The CCG approach trades lower memory consumption for increased computation, thus the parallelization provides advantages with respect to the available memory sizes and the parallel computation.



(a) Connected process graphs.



(b) Distributed process graphs.

Figure 4.23: Examples of connected (a) and distributed (b) CCGs. When splitting the former into three partitions, some redundant nodes (blue) need to be (re-)introduced.

4.7.1 Distributed Data Decomposition

In general, the key to efficient parallelization of data intensive problems lies in a suitable domain decomposition for the data [KDS99]. Splitting a parallel trace graph into single process graphs is the first and most convenient way for decomposition, compare Figure 4.23. Then construction and evaluation of process graphs can be done in parallel and independently, see Sections 4.1.1 and 4.7.5.

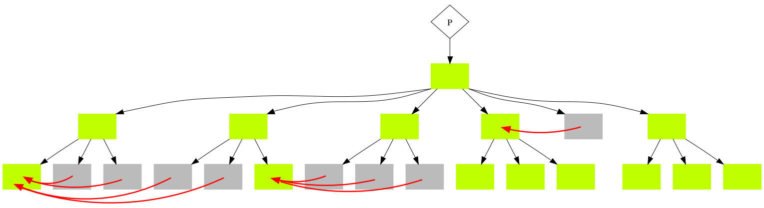
If decomposition into single process graphs is insufficient, splitting of single process graphs is inevitable and maximum sub-graphs (sub-trees) of a process graph need to be distributed to remote locations. This scheme produces a number of remote sub-graphs and an incomplete local root tree of the original graphincluding so called *stub nodes*, that refer to a remote location instead of a local sub-graph, see Figure 4.24(b). Different partitioning schemes may result in different separated sub-graphs, compare Figure 4.24(b,c) and Section 4.7.3.

Remote references are allowed only from stub nodes in the local root graph to the root nodes of the remote sub-graphs but not between remote sub-graphs. This will guarantee separate, unconnected remote sub-graphs which will be an advantage for evaluation, see Section 4.7.5 below. Due to this restriction, distributed CCGs will re-introduce redundant graph nodes across the partitions, that could be removed with non-distributed compression, compare also Section 4.7.3.

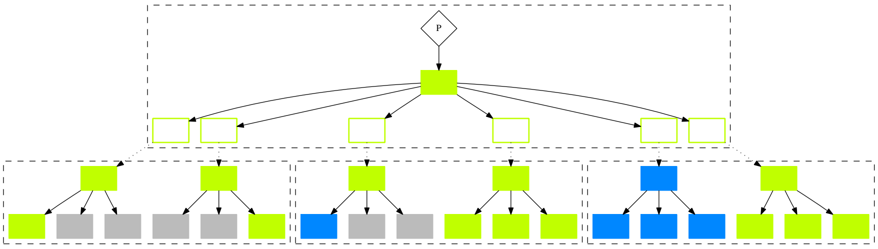
4.7.2 Distributed CCG Construction

The proposed distributed CCG construction algorithm uses *active partners* that perform construction and compression as well as *passive partners* that serve as a mere storage locations. It allows to handle t parallel process traces with $p \leq t$ active entities. If this is insufficient, data can be moved to $q \geq t$ passive storage locations.

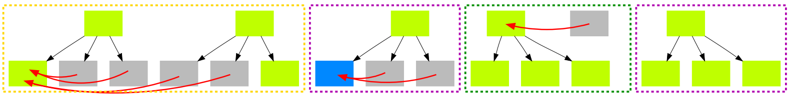
Every process graph is read sequentially by one of the active entities, creating a local graph, compare Section 4.1.1. As soon as a certain memory consumption is reached, one or more sub-graphs are moved to a *passive* remote location. This has to include all previously finalized nodes, otherwise it would be impossible to separate nodes from either sub-tree at a later time.



(a) Single connected process graph.



(b) Process graph with distributed sub-graphs.



(c) Alternative partitioning of the sub-graphs.

Figure 4.24: Decomposition of a single connected process graph (a) into distributed sub-graphs (b) with three partitions and an incomplete root tree with stub nodes (empty rectangles). An alternative partitioning (c) shows that the number of redundant nodes (blue) changes notably.

Construction and compression continue with an empty supply of replacement candidates and reduced local memory consumption. In the end, the active entities will keep the remaining sub-trees as well as the incomplete local root trees of every trace process. Moving sub-graphs between remote storage locations can be implemented by means of serialization and restore including re-referencing, compare Section 4.6. As an alternative, the control of CCG construction for a process trace can be passed to another location that becomes the current active entity. This could avoid the transfer of extensive sub-trees, yet it still enforces a serial processing scheme, where every process traces is handled by a single active entity at a time.

4.7.3 Distributed CCG Compression

Distributed storage is suitable to handle very large CCGs that would not fit to main memory of a single computing node even in compressed form. Both, storage size and computational effort can be split to a fraction of the original value. However, distribution is not transparent with respect to the compression ratios R_{nodes} and R_{memory}.

In general, compression ratios decrease if the distributed scheme of CCG compression is applied, because remote compatible nodes cannot be replaced by references but must remain redundantly. In the worst case, distribution into p partitions re-produces every node p times. This means, node count and memory consumption grow by factor p and the compression ratios $R_{nodes} \geq p$ and $R_{memory} \geq p$ are reduced to R_{nodes}/p and R_{memory}/p.

In the best case, there are no remote redundancies. This does not necessarily mean that all pairwise compatible nodes are located in the same partition. It is sufficient, that for every node A there is a compatible local node B even if there are other remote compatible nodes C. In this case, the only remaining overhead consists of n_{stub} stub nodes where $n_{stub} \geq p$ is the granularity of the distribution.

Assumed the total node count $N \gg n_{stub}$ is much larger than the granularity, the compression ratios R_{nodes} and R_{memory} remain almost unchanged.

In practice, the reduction of compression ratios will be near best case it is improbable that a node could be referenced from *all* remote locations but *not at all* locally.

On average, distributing a large CCG to p separated smaller CCGs will have the same effect on compression ratios as compressing p independent CCGs of smaller size, including the trend that larger CCGs (the non-distributed one) usually yield higher compression ratios than smaller ones (the distributed partitions), compare the compression model in Section 5.1.1.

The Compression speed is affected by the distributed approach in two ways. Assume a sufficiently balanced distribution, which is assumed to reflect the average case. On one hand, the compression effort decreases inverse proportional to the number of active parallel entities $p \leq t$ because each needs to process N/p nodes instead of N. On the other hand, the search for replacement nodes needs to traverse a potentially shorter candidate list, because only a sub-set of all global candidates is present at the same location. The latter effect is almost completely ceased if a limited search strategy is applied, compare Section 4.2.1. Both effects combined, produce a superlinear speed-up for $p \leq t$ parallel entities. However, the speed-up cannot grow further than t when distributing sub-graphs to $q \geq t$ locations, as every process trace is read linearly.

4.7.4 Distributed Serialization and Restoring

Distributed serialization and restoring can be achieved more or less in the same way as discussed in Section 4.6. A few minor issues allow further flexibility for serialized distributed CCGs.

As a precondition, the distributed parts of a CCG should be serialized individually using a disjoint naming or numbering scheme except for stub nodes which should be named identical to the nodes they refer to. This would allow to unite multiple parts of a distributed CCG by simply concatenating the respective serialized files. Then, restoring of a distributed CCG is possible with original or reduced granularity.

4.7.5 Distributed Evaluation

Since distribution of CCGs is data-driven rather than computation-driven, the associated evaluation algorithms should adapt to the distributed data.

For the summary query algorithm (Section 4.5.3) an efficient distributed implementation works like following: At first, the unchanged algorithm is applied to all partitions separately, returning a partial result each. Then, the global result is summarized from all collected partial results. Assuming the effort for subsuming partial results is insignificant, the parallel speed-up of this scheme is close to p with p distributed partitions of balanced size.

For other sequential evaluation algorithms, there are no convenient and efficient distributed counterparts, for example for the iterator algorithms or the send-receive matching algorithms. The general difficulties of distributed event trace analysis are not specific to the CCG data structure but apply to distributed versions of traditional data structures in the same way, compare for example [Bru08, GWWM06, GKP^{+}07].

5 Evaluation of CCG Algorithms

This chapter presents theoretical and practical evaluation of construction and compression as well as querying for the Complete Call Graph data structure. The practical part contains experiment results for synthetic examples to investigate the worst and best case scenarios, on one hand, and for real-world application examples to show realistic behavior, on the other hand. Finally, recommendations for various parameters are derived. Furthermore, the results are compared to the state-of-the-art in this area.

5.1 Theoretical and Synthetic Evaluation

At first, the general compression behavior and the maximum spectrum of compression ratios is illustrated with theoretical models and synthetic experiment results.

5.1.1 Compression Model

Here, a simplified compression model is designed for node compression ration R_{nodes} with respect to a single deviation bound p. Let $g^1, ..., g^N$ be N graph nodes with identical hard properties. Let each node g^i have d soft properties $v_1^i, ..., v_d^i$. Assume N is sufficiently large and the values v_j^i are statistically independent and bounded by finite limits.

Two nodes g^a and g^b are compatible if $\left|v_i^a - v_i^b\right| \leq p$, with a common deviation bound p. Each node g^i can be mapped to position $(v_j)_{j=1,...,d}$ in a d-dimensional parameter space. There it covers the d-dimensional area

$$A_p^i = \prod_j \left[p_j^i - p, p_j^i + p\right], \tag{5.1}$$

that means it is would be suitable as replacement for any other node mapped inside A_p^i. The d-dimensional volume $a_p = \left|A_p^i\right|$ is $a_p = (2p)^d$ independent of i. Let furthermore L be the (average) diameter in every dimension of the d-dimensional cube C containing all v^i.

In order to cover C completely according to d-dimensional volume n nodes with

$$n \geq \frac{L^d}{a_p} = \frac{L^d}{(2p)^d}, \quad n \geq 1 \tag{5.2}$$

are necessary but at least one. It follows

$$n \geq \max\left(1, \left(\frac{L}{2p}\right)^d\right). \tag{5.3}$$

With the definition of the node compression ratio it follows

$$R_{nodes} := \frac{N}{n} \leq \frac{N}{max\left(1, \left(\frac{L}{2p}\right)^d\right)} \leq N. \tag{5.4}$$

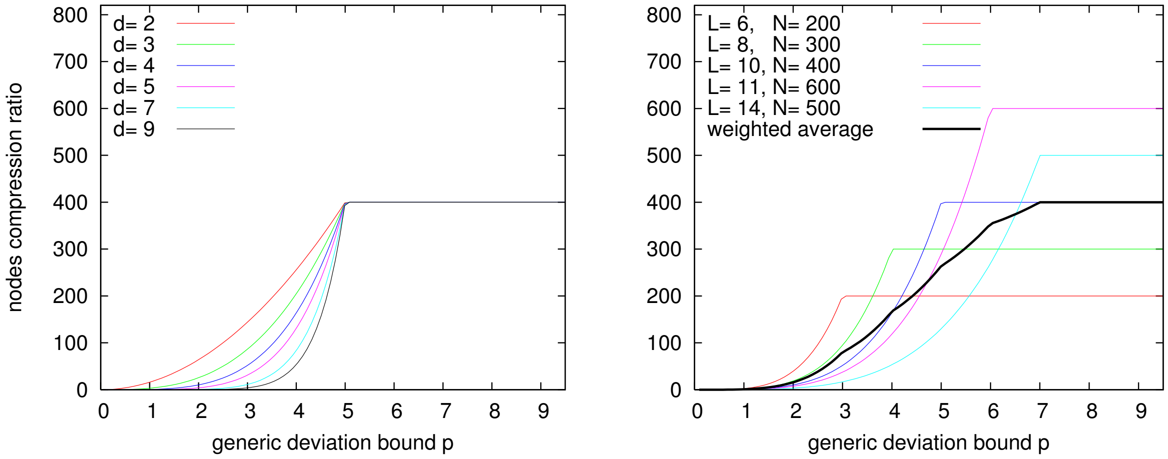


Figure 5.1: CCG compression model: R_{nodes} vs. deviation bound for varying dimension (number of soft properties per node) $d = 2, ...9$ with $N = 400$ and $L = 10$ (left) and for varying node count N and diameter L with $d = 4$ (right).

Figure 5.1(left) shows an example of the theoretical limit for the node compression ratio R_{nodes} according to Equation (5.4). The edge at $p = L/2 = 5$ marks the point where a single node would be sufficient to cover C. Then the maximum compression ratio N is reached. Therefore, the interval $0 \leq p \leq L/2$ is the more interesting one.

Furthermore, it is assumed that L and N are independent. That means, after sufficiently many nodes of a certain kind the d-dimensional cube C will not grow anymore when increasing N further. Thus, the diameter L will stay constant. Following this reasoning the compression ratio is predicted to grow for larger traces of the same kind.

All real-world traces will composed of many sets of nodes with identical hard properties. The set may differ with respect to the number of soft properties d, the size of the set N and the d-dimensional diameter L. Therefore the over-all compression ratio will be combined as a weighted average from many separate R^x_{nodes}. Figure 5.1(right) shows an example of a number of separate compression models according to Equation (5.4) and their weighted average.

5.1.2 Non-Monotone Compression

Contrary to the previous model, CCG compression is not strictly monotonous with respect to any soft property p_*. Although, larger values for p_* will typically provide better compression, this is not necessarily so. Figure 5.2 shows an example of the (unlikely) case where increased deviation bounds lead to less compression. In Figure 5.2(top) function `foo` calls `bar` five times. All occurrences of `barX` are compatible with respect to hard properties. With respect to soft properties p_i and deviation bound ± 3 the second occurrence `bar2` is compatible to the first one `bar1`. Therefore, node `bar2` is replaced by a reference to `bar1`. All other nodes are not compatible to each other or to `bar1`. The resulting nodes compression ratio is $R_{nodes} = \frac{6}{5}$.

Now assume that with smaller compression parameters ± 2 the nodes `bar1` and `bar2` are no longer compatible. Then `bar2` is not removed but still available. Because node compatibility is not a transitive relation it is possible that `bar3`, `bar4` and `bar5` can be replaced by references to `bar2` like shown in 5.2(bottom). The node compression ratio for the latter case is $R_{nodes} = \frac{6}{3}$. Thus, smaller compression parameters may lead to better compression, q.e.d.

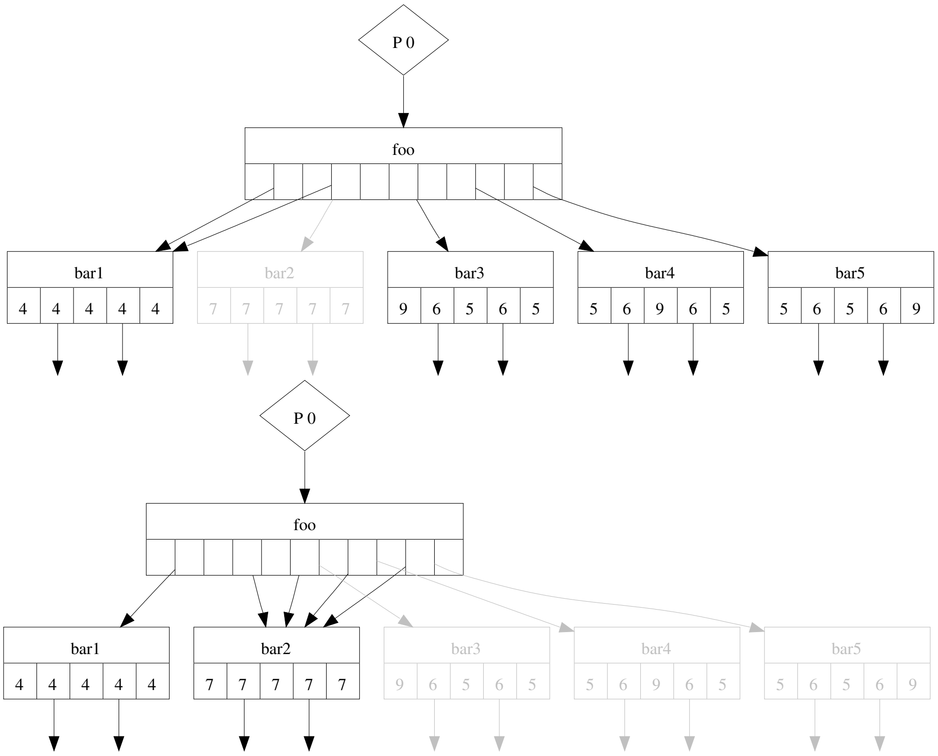


Figure 5.2: Example of an compressed CCG that achieves less compression with the generic bound ± 3 (top) and *improved* compression with the *reduced* deviation bound ± 2 (bottom).

The reason is that in general, node replacement is not chosen optimally but according to a heuristic. Smaller compression parameters may inhibit compression of some nodes earlier. Later on, the existence of that particular node might induce better compression that would be impossible otherwise.

This counter example is no contradiction to the previous monotonous compression model from Section 5.1.1. Rather, the monotonous compression model is valid for large numbers of nodes whereas non-monotone compression is happening with few nodes only. With more nodes it is probable that another replacement will neutralize the penalty of a case of non-optimal compression.

5.1.3 Best Case Compression

Best case experiments examine the suitability of CCG compression for very regular traces with steady timing behavior. Firstly, it is tested with the simplest synthetic trace that contains a long sequence of calls to the same function. Figure 5.3 shows the number of required graph nodes versus the number of enter/leave events in the trace: For up to 20 million events there are only 7 to 26 graph nodes required, depending on the branching factor b. This behavior clearly reflects a logarithmic growth rounded to integers. This relates to the tree depth that would be necessary to store all nodes in a balanced tree.

In this example the maximum compression is as big as $R_{nodes} > 2\,850\,000$ and $R_{memory} > 800\,000$. With linear growth of the event count and logarithmic growth of CCG node count, the asymptotic compression ratio is infinitely large. Another example uses a slightly less regular function call pattern including deep recursion. Figure 5.4 gives the CCG node count versus the event count for an unbalanced binary recursion call pattern. Such patterns typically occur in *divide-and-conquer* algorithms like quicksort or, like in this synthetic example, the trivial recursive computation of the Fibonacci sequence.

This example needs only few more nodes than the example in Figure 5.3, showing the same logarithmic growth. Therefore, the asymptotic compression grows infinitely as well. The maximum branching factor b has no influence here, because the *effective* branching factor is determined by the *binary* recursion.

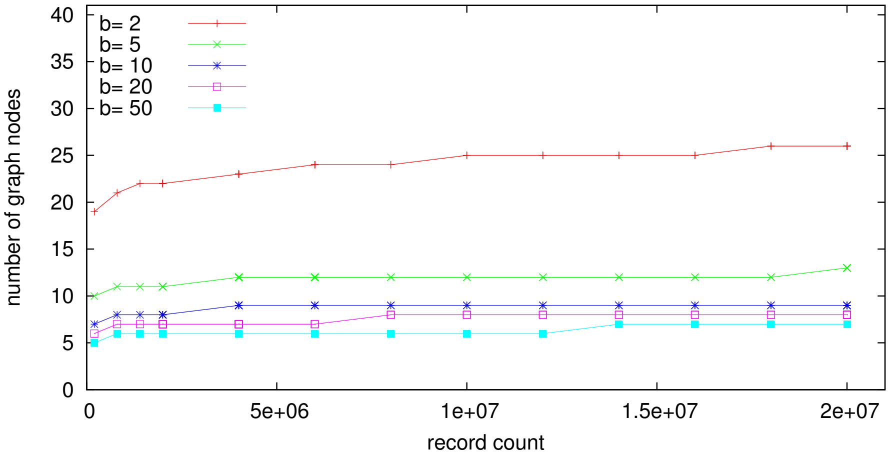


Figure 5.3: The CCG node count n achieved with a most regular call structure. (The corresponding node compression ratio R_{nodes} ranges from $10\,500$ to $2\,850\,000$!)

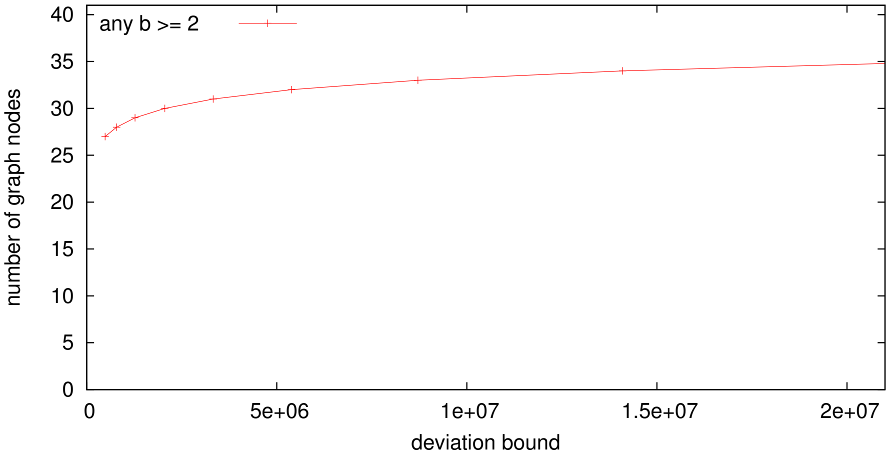


Figure 5.4: The CCG node count n for an unbalanced binary recursion call pattern. (The corresponding node compression ratio R_{nodes} ranges from $17\,000$ to $600\,000$!)

5.1.4 Worst Case Compression

After the best-case evaluation, the other end of the spectrum shall be examined with the most difficult examples for compression. This will involve two scenarios: Firstly, most irregular function call patterns and secondly, very irregular timing behavior. This are the most typical cases for irregularities in hard properties or soft properties and serve as an example for other cases.

The example for very irregular call patterns is a long sequence of calls to randomly selected functions out of 10 candidates with regular timing. With branching factor b this results in 10^b possible leaf nodes that are pairwise incompatible with respect to hard properties. The number of possible artificial sub-trees formed from this will grow even more tremendously due to the combinatorial complexity. This resembles the worst-case situation for deeper function call patterns including sub-calls, sub-sub-calls, etc.

Figure 5.5 shows the resulting R_{nodes} which is almost constant for growing record count, except for the outlier $b = 5$. R_{memory} which is not shown behaves similarly. Concluding from this experiments, CCG compression is able to achieve substantial compression for very irregular call patterns with sufficient event count. As anticipated, the compression ratios are much smaller than for regular examples.

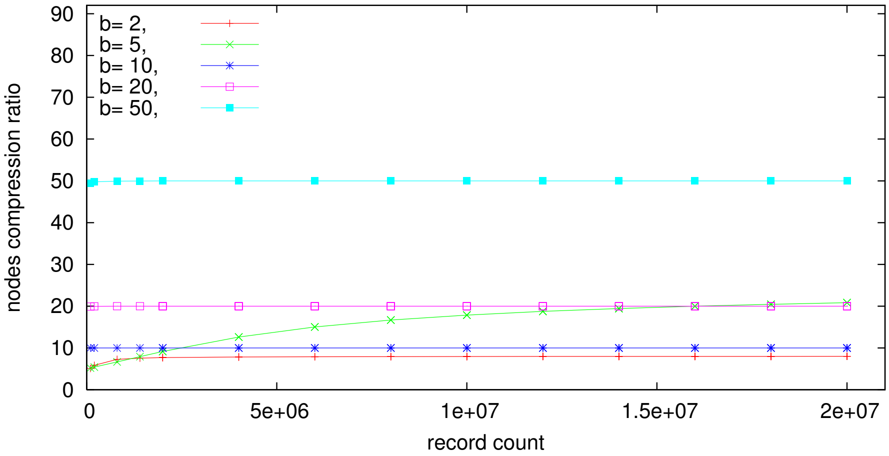


Figure 5.5: Node compression ratio for random call patterns with regular timing.

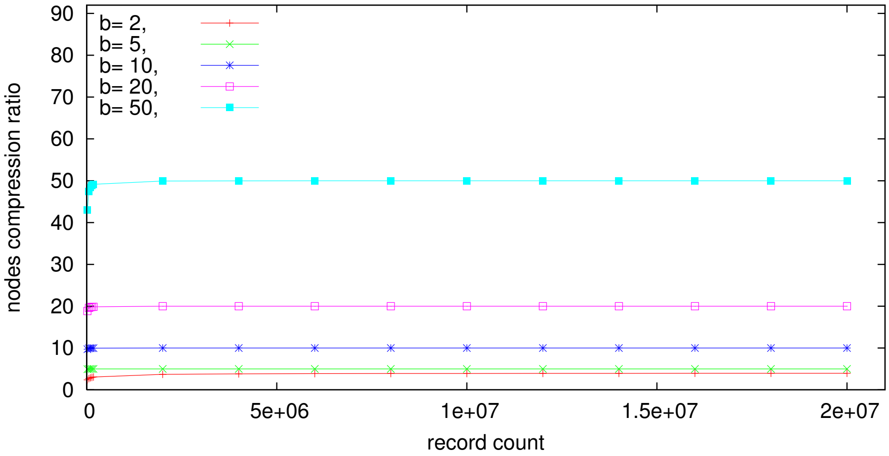


Figure 5.6: Node compression ratio for regular call patterns with random timing and small deviation bounds for timing ($abs = 1000$, $rel = 10\%$).

The experiments investigating the impact of irregularity in soft properties are sensitive to the deviation bounds, which causes diverse behavior. Now, the function call pattern is regular again. At first, small deviation bounds ($abs = 1000$, $rel = 10\%$) are used, which lie below the variability of the particular soft properties. The resulting compression shown in Figure 5.6 is very similar to the previous experiment's results in Figure 5.5 because it creates a similar random scheme of pairwise (in)compatible nodes.

As deviation bounds nearly cover the complete range of the random property, the compression ability improves notably, like shown in Figure 5.7 with larger bounds for time deviation (abs=10000, rel=100%). The improvement is obviously favoring small branching factors b, because with less values per node it is more likely that all are covered by the deviation bounds. As soon as the deviation bounds are large enough to cover all samples always, it will approach the behavior of the best case, see Section 5.1.3.

In combination, irregularity in the call pattern and in timing, i.e. in hard and soft properties, decreases the compression further, as expected. Figure 5.8 shows the results for *random* call patterns and *random* timing with very small deviation bounds (abs=10, rel=1%). Still, all R_{nodes} are above 2. Larger branching factors b show an advantage, after a longer saturation phase after which the compression ratios remain on an almost constant level. R_{memory} behaves similar to R_{nodes} but on lower levels between 2 and 7.5.

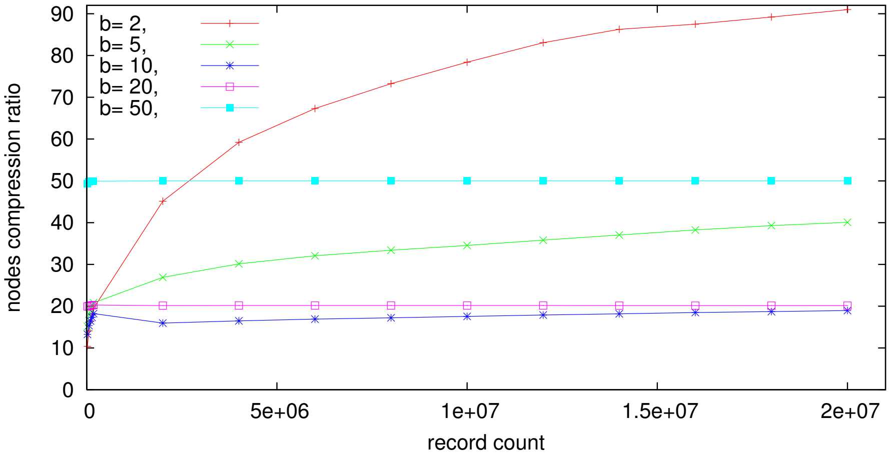


Figure 5.7: Node compression ratio for regular call pattern with random timing and large deviation bounds for timing ($abs = 10000$, $rel = 100\%$).

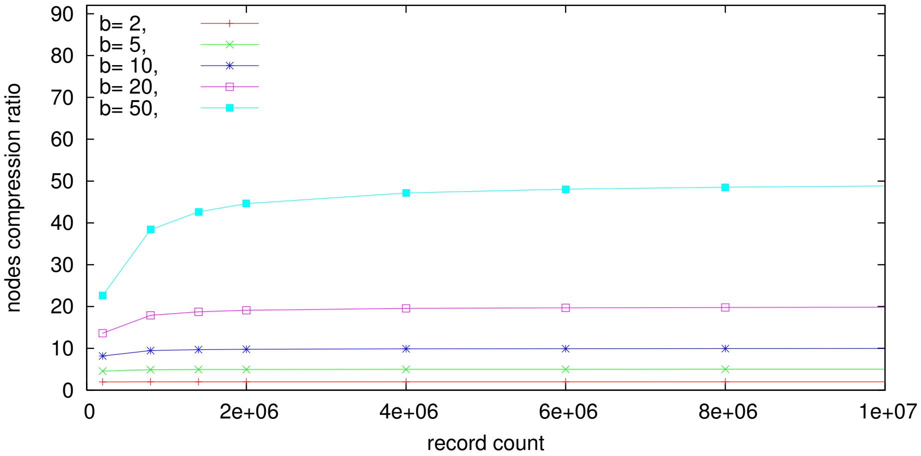


Figure 5.8: Node compression ratio R_{nodes} for *random* call pattern combined with random timing and very small deviation bounds (abs=10, rel=1%).

5.2 Real-World Construction and Compression

The following experiments examine the real-world compression behavior of CCG construction and compression. As example application the *SMG2000* code is used [JMC02, Fal00, BFJ00]. It is a parallel semicoarsening multigrid solver for a special kind of linear systems with very good scalability. *SMG2000* is a widely known benchmark code and is available as part of the *ASCI Purple* benchmark suite[1].

All experiments use an example trace from *SMG2000* with 8 CPUs and approximately 36 million event records. It was created on a cluster of dual Intel Xeon nodes with 3 GHz, 512 KB cache and 2 GB main memory memory per node. The timer resolution is identical to the CPU clock tick of 0.33 ns. In OTF format [KBB08] the trace accounts for 184 MB of compressed trace file size (563 MB uncompressed).

The CCG experiments were performed on a cluster of dual AMD Opteron nodes with 2.2 GHz, 1024 KB cache and 4 GB main memory per node. Each of the CCG construction and compression runs was strictly sequential, though. Distributed computation was only used for the extensive parameter studies.

---

[1] ASCI: Accelerated Strategic Computing Initiative, USA supercomputing initiative that started in 1992. Nowcalled ASC: Advanced Simulation and Computing Program.

The experimental evaluation of CCG Construction and Compression focuses on two main characteristics: Firstly, the degree of compression, and secondly, the compression effort respectively compression run-time. It will investigate several influencing factors for both:

- deviation bounds for soft properties,
- branching factor b,
- event count, i.e. trace size, and
- maximum node search length.

The most important factors are the deviation bounds for soft properties that may also interact with one another. The influence of deviation bounds is presented based on experiment results for the two most common and most important soft properties: the absolute and relative time deviations. This is divided in two typical application scenarios for small scale and large scale compression. When using two or more soft properties, it is safe to assume that either all are kept quite restricted or all are rather relaxed. Opposing settings for coexisting soft properties seem unusual.

The following factors are investigated with respect to their effect on small scale and large scale compression but not with respect to their mutual interaction. Recommendations for parameter setting based on experiment result will be postponed until Section 5.6. Then the results of CCG construction and compression will be balanced against corresponding results for query operations.

5.2.1 Small Scale Compression

Small scale compression covers the cases where deviations due to lossy compression are small or non existent. Here, absolute time deviation is allowed $abs = 1 \ldots 10000\,\mathrm{ticks}$ which equals $0.33\,ns$ to $3.33\,\mu s$ in this example with a 3 GHz CPU and relative deviation ranges over $rel = 0.001 \ldots 1.0$ (0.1% to 100%). Those values allow notable compression while preserving reasonable accuracy with respect to temporal behavior. Furthermore, the results for lossless compression are shown ($abs = 0$, $rel = 0.0$).

The node compression ratio R_{nodes} ranges from 5 to 55 for the given compression parameters abs and rel, see Figures 5.9 and 5.12. The former equals the result for lossless compression, the latter is achieved with $abs = 10000$ and $rel = 1.0$. The growth process with respect to one compression parameter abs or rel looks monotonous. It saturates when the other compression parameter prevents further compression. This meets the theoretical model, compare Section 5.1.1.

The memory compression ratio R_{memory} behaves similarly but on a lower level, see Figures 5.10 and 5.13. For lossless compression or very small deviation bounds $R_{memory} = 3$ is produced. It is growing up to $R_{memory} = 17$ in the current scope (for $abs = 10000$, $rel = 1.0$). The similarity of the behavior of R_{nodes} and R_{memory} is typical. The graphs look almost identical except for a factor f with $1 < f < 4$. Although in general R_{nodes} and R_{memory} are not strictly proportional (see Section 4.2.5) they behave similar for a constant b. This similarity will reappear in following experiments. Therefore sometimes only one graph will be shown.

Compression run-time for the above experiments ranges from $6560\,s$ ($1.8\,h!$) for lossless compression to $4700\,s$ ($1.3\,h!$) for very small scale compression to $65\,s$ for medium scale compression (Figure 5.11). Most experiments show a reasonable run-time, only cases with very small deviation bounds $abs \leq 100$ or $rel \leq 0.01$ are extremely slow. For larger bounds the run-time is more or less constant.

As the compression run-time is dominated by search for replacement nodes the plot of run-time vs. node count in Figure 5.14 allows better insight in the cause of very slow compression. The search operation consists of two stages. First, a hash look-up provides a candidate list with constant effort $O(1)$. Then, those lists need to be traversed with linear effort $O(l)$ with respect to the average list length l, compare Section 4.2.1. With small scale compression and large node count this results in quadratic effort $O(l^2)$ because for every list entry a complete list traversal is performed. This is clearly visible as worst case behavior in Figure 5.14.

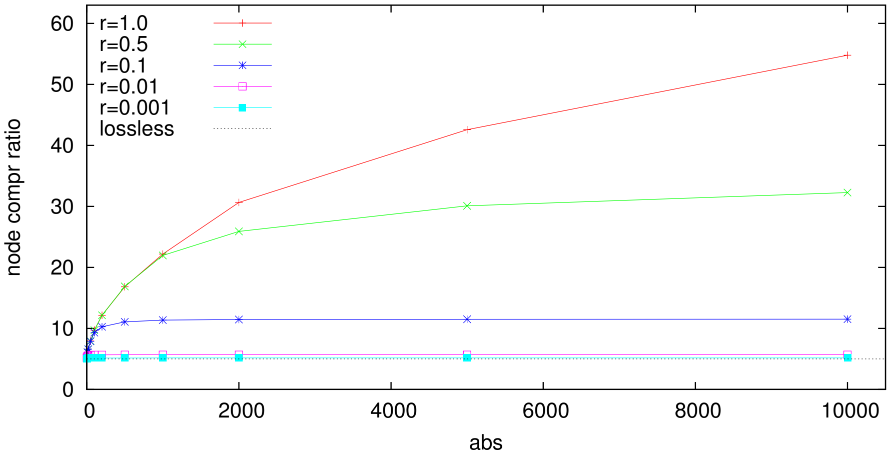


Figure 5.9: Node compression ratio R_{nodes} vs. time deviation abs for small scale compression.

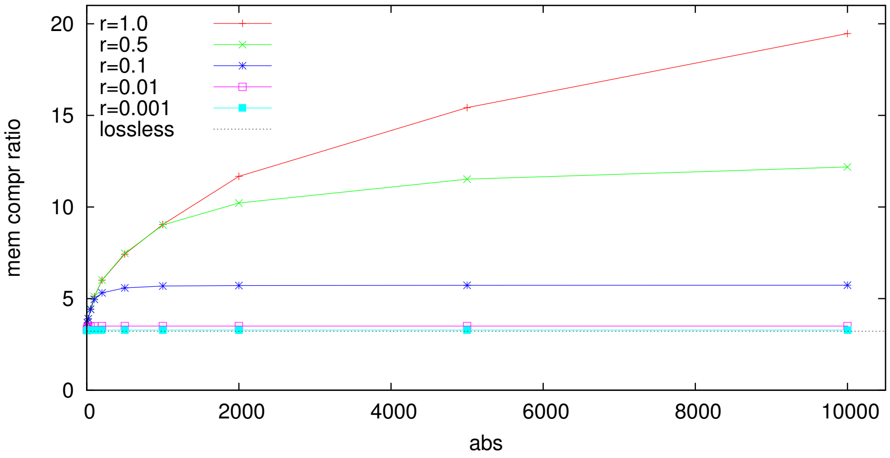


Figure 5.10: Memory compression ratio R_{memory} vs. time deviation abs for small scale compression.

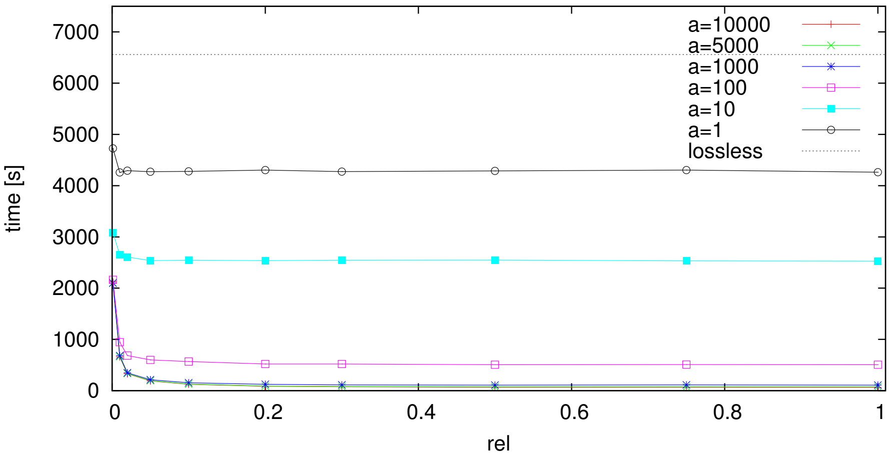


Figure 5.11: Compression run-time for small scale compression vs. time deviation rel.

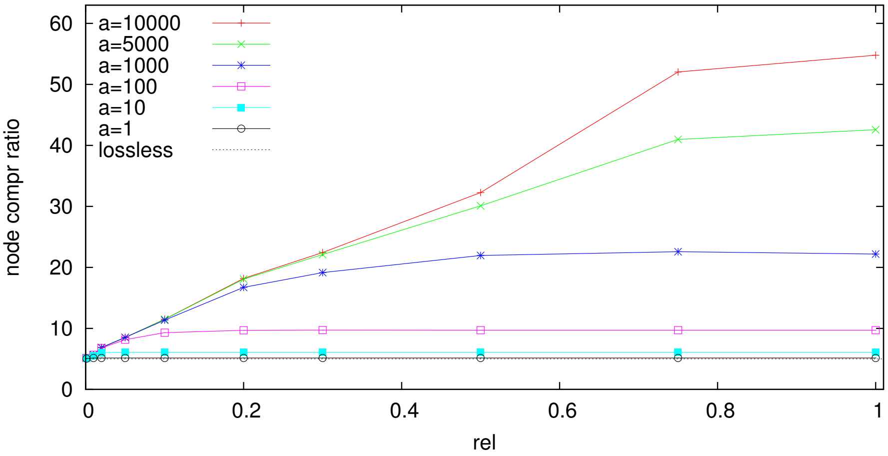


Figure 5.12: Node compression ratio R_{nodes} vs. time deviation rel for small scale compression.

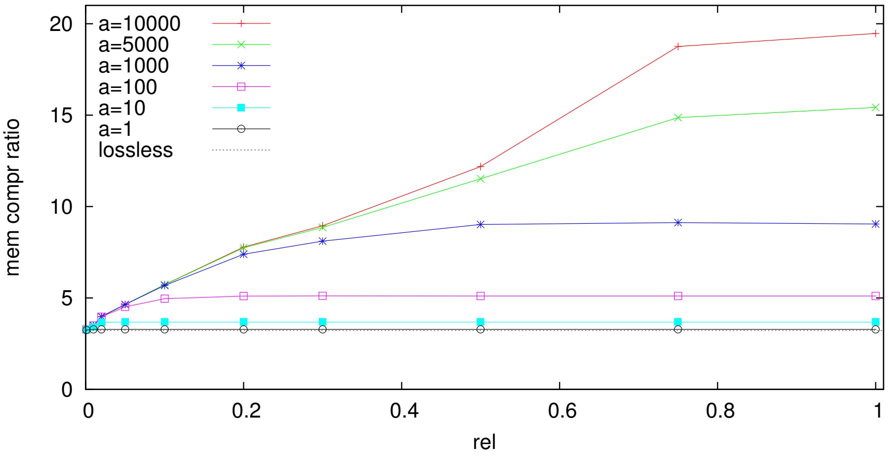


Figure 5.13: Memory compression ratio R_{memory} vs. time deviation rel for small scale compression.

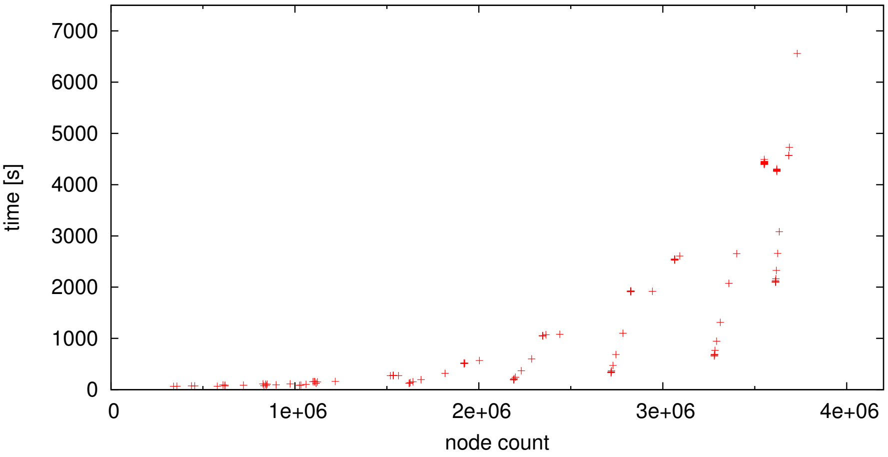


Figure 5.14: Influence of final node count n on compression run-time for small scale compression.

5.2.2 Large Scale Compression

Large scale compression allows deviation bounds of $abs = 10\,000$ ... $10\,000\,000$ ($3.33\,\mu s$ to $3.33\,ms$ in this example) and $rel = 0.1$ ... 10.0 ($10\,\%$ to $1\,000\,\%$). Therefore, it achieves much better compression and is much faster than small scale compression in the previous experiments.

The node compression ratio varies from $R_{nodes} = 11$ to 649, see Figures 5.15 and 5.18. With respect to the abs parameter R_{nodes} grows like predicted in Section 5.1.1. With respect to rel the initial growth is inhibited (for small $rel < 0.5$). For larger $rel > 0.5$ it follows the normal behavior. This complies with the theoretical model in Section 5.1.1 because both compression parameters, abs and rel influence the compression concurrently. In this case, the stricter one determines the over-all compression result.

For comparison the maximum possible compression ratio $R^{max}_{nodes} = 1112$ is given as well. It is achieved with unlimited compression parameters. Maximum possible compression is determined by the CCG itself respecting only hard properties but no soft properties.

Like before, the behavior of R_{memory} is similar to R_{nodes}. It is ranging from 5 to 220, see Figures 5.16 and 5.19. The maximum memory compression ratio reaches $R^{max}_{memory} = 458$.

The compression run-time behavior for large scale compression is notably faster than small scale compression in previous section. Now, time ranges from $\approx 48\,s$ to $\approx 125\,s$ (Figure 5.17). Again, there is an increase in run-time towards small compression parameters $abs \leq 10\,000$ and $rel < 1.0$ but not as excessively as in the small scale case. Apart from that run-time is constant below $60\,s$. It is very close to the lower bound for compression run-time of $48\,s$ which is achieved with maximum compression.

The run-time depending on node count (Figure 5.20) reveals a notable difference compared to small scale compression (Figure 5.14). Obviously, the maximum effort is not quadratic anymore but linear $O(l)$, i.e. the search effort for replacements per node seems constant. This is possible because the distribution of nodes changes from small scale compression to large scale one.

The reason can be found in the two-stage data structure keeping replacement candidates, compare Section 4.2.1. In its second stage there are linear lists of nodes with identical hard properties. For every replacement candidate one of those lists is traversed linearly. For small scale compression, those lists usually grow rather long. Now, with large scale compression, the length of those lists is generally quite short because few nodes are sufficient to represent all following ones. Following experiments will demonstrate, that restricting the linear search to a limited number of nodes will decrease the over-all effort notably with minimal decrease in compression. See Section 5.2.5, compare also Sections 4.2.1 and 4.2.2.

The correlation of higher compression with reduced effort may seem surprising at first. However, one must not conclude that large scale compression is always superior to small scale compression because it also influences the accuracy of all soft properties.

Corollary 1. *For CCG compression the following correlation can be observed on average: Increased deviation bounds for soft properties, which reduce the accuracy of the compressed representation, yield better compression ratios and reduce the computational effort.*

5.2.3 Influence of Branching Factor

The maximum branching factor b is a basic graph property which affects both, uncompressed and compressed CCGs. Since the original branching factors of call trees are usually unbounded, nodes with more than b child nodes are split into multiple so called *artificial nodes* with $\leq b$ children each, see Sections 3.2.2 and 4.1.2. This is done before actual compression. Therefore, the absolute node count and the absolute memory consumption are influenced by this parameter. Both behave similar in uncompressed mode and in various compression modes from lossless and small scale compression to large scale and unlimited compression. Figure 5.21 shows the influence of b to the node count n. Always, there is an almost constant behavior for $b \geq 10$ and a distinct overhead for $b < 10$ which is rising for $b \rightarrow 2$. The reasons for this is obvious, because small b require more original nodes to be split into a larger number of small nodes, thus raising the node count.

The influence of b to the memory consumption m (not shown) looks very similar to the one shown in Figure 5.21 for all cases with a range from 2.3 MB to 1570 MB. The explanation is similar as for n above. Indeed, the memory consumption for more small nodes are partly compensated because the memory consumption per node decreases. Yet, there is additional memory consumption for references (pointers) to nodes which increases for larger node counts.

Figure 5.22 shows the effect of b to R_{nodes} and R_{memory}. In general, a higher branching factor allows better node compression (R_{nodes}). There is a strong increase for small $b \leq 20$ and only a moderate further growth for larger b. The behavior of the memory compression ratio R_{memory} differs notably from that for R_{nodes}. For small b there is a small growth of R_{memory}, too. Yet, for larger b and large scale compression there is a slow decline of R_{memory}. For small scale compression ($abs \leq 100$, $rel \leq 0.1$) R_{memory} is almost constant not declining.

The opposite effects emerge from the quotient of uncompressed and compressed node count or memory usage. In fact, all absolute values N, n, M and m show an initial steep decline followed by an almost constant behavior, like shown in Figure 5.21 for n and N.

Corollary 2. *For growing branching factors and sufficiently large compression the compression ratios* R_{nodes} *and* R_{memory} *show opposite behavior. At the same time, the node counts* N *(total) and* n *(compressed) and the memory consumption* M *(total) and* m *(compressed) stay almost constant for* $b \geq 10$.

Therefore, sufficiently large values of $b \geq 20$ are recommended with respect to actual resource usage n and m. Larger b provide no additional advantage. Considering the preferences of CCG queries (see Section 5.4) which profit from small branching factors the over-all recommendation will be a value of $b = 20$. This is also used in all experiments, unless stated otherwise.

Apart from the compression ability, run-time is very important, see Figure 5.23. For $b < 10$ there is a small disadvantage in run-time whereas for $b \geq 10$ it is almost constant. This supports above recommendation of $b = 20$.

5.2.4 Influence of Trace Size

The size of input traces is not a free parameter, i.e. it cannot be freely adjusted. Yet, it is a fundamental influence to the storage data structure. According to the theoretical compression model the *maximum* possible compression grows with the number of input records, i.e. with the uncompressed node count N, see Section 5.1.1. The following experiments will investigate the compression behavior in practice.

The first of two experiments compares the compression behavior for a set of related parallel traces (on horizontal axis) with different compression parameters (abs, rel). All trace originate from the same application (again SMG2000) runnig on the same platform with varying degree of parallelization and different problem sizes. All other input parameters (solver, domain decomposition, blocking) were unchanged. The resulting traces are expected to show same *general behavior* and the same degree of *general regularity or irregularity* (without attempting precise definitions of this terms).

Figure 5.24 shows the result of the first experiment. For all parameter sets (abs, rel) the compression ratios R_{nodes} stay broadly constant for all the example traces with varying record count. There are notable fluctuations for some traces (e.g. at 12.5 M records). This presumably caused by slightly anomalous traces, because the extend of the fluctuations is proportional to the value of R_{nodes}. Yet, the outliers do not obscure the obvious general trend.

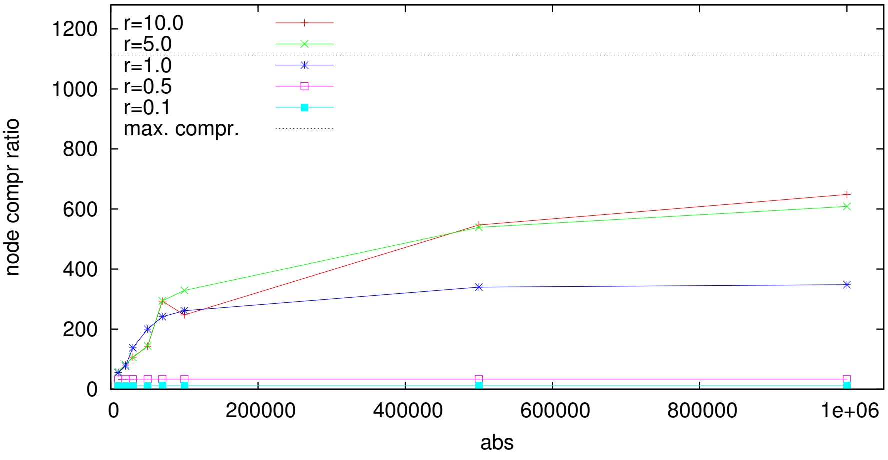


Figure 5.15: Node compression ratio R_{nodes} vs. time deviation abs for large scale compression.

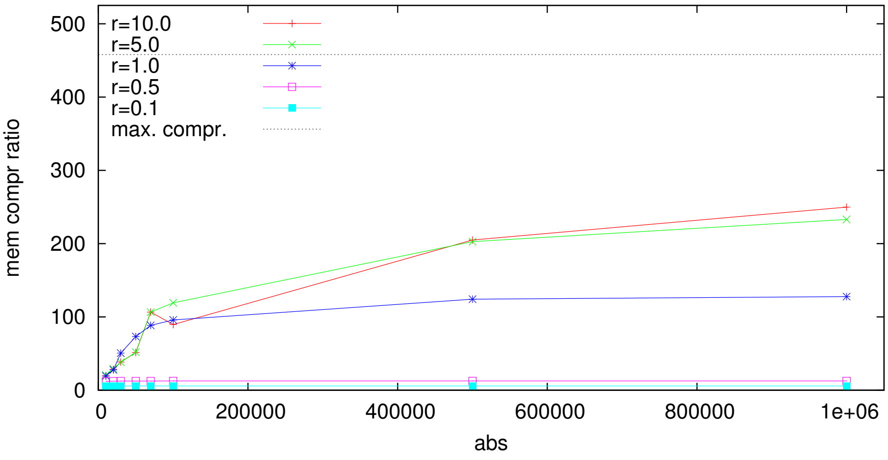


Figure 5.16: Memory compression ratio R_{memory} vs. time deviation abs for large scale compression.

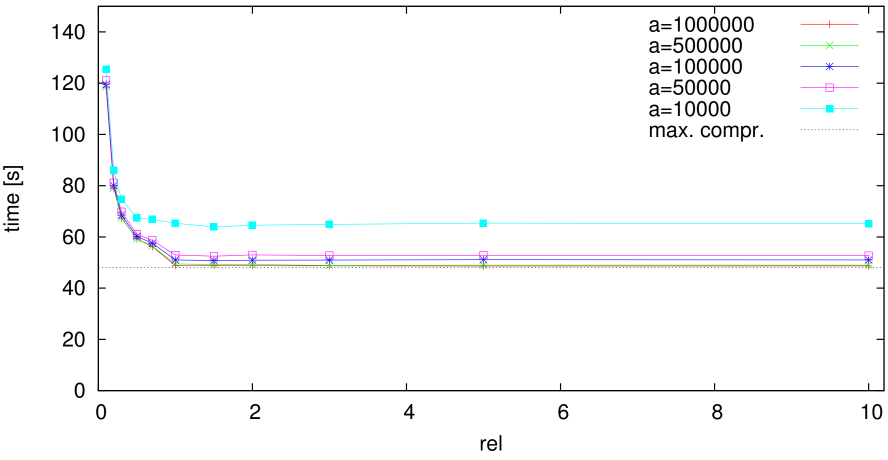


Figure 5.17: Compression run-time for large scale compression vs. time deviation rel.

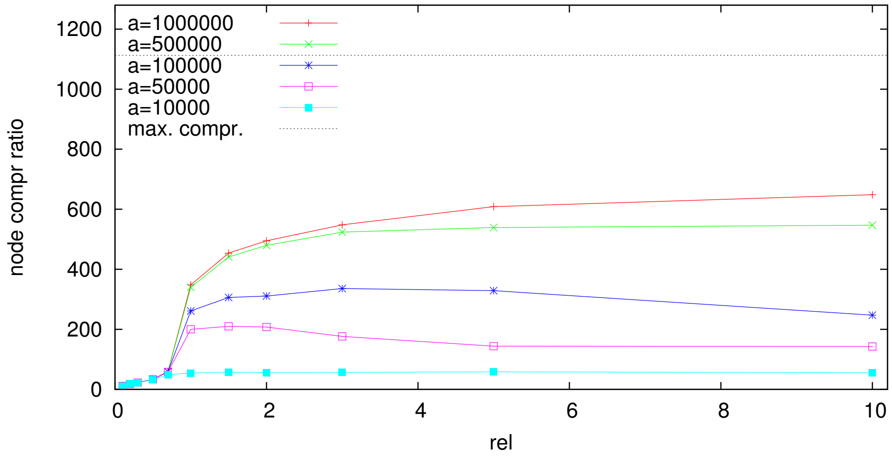


Figure 5.18: Node compression ratio R_{nodes} vs. time deviation rel for large scale compression.

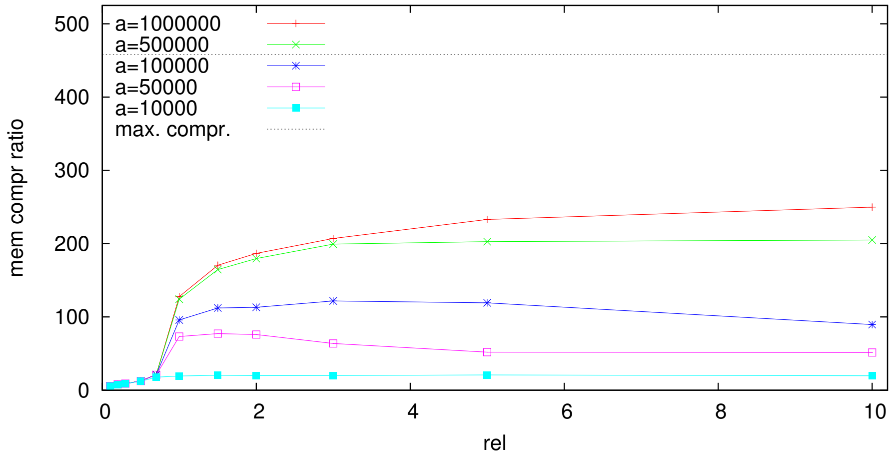


Figure 5.19: Memory compression ratio R_{memory} vs. time deviation rel for large scale compression.

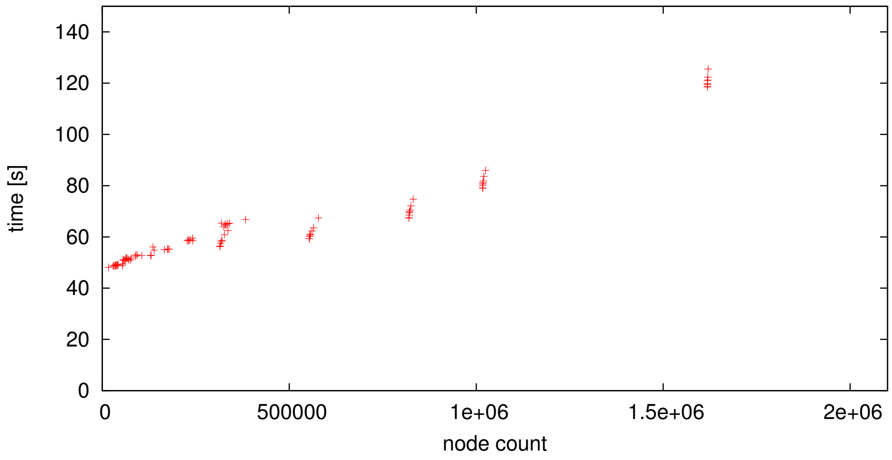


Figure 5.20: Influence of final node count n on compression run-time for large sclale compression.

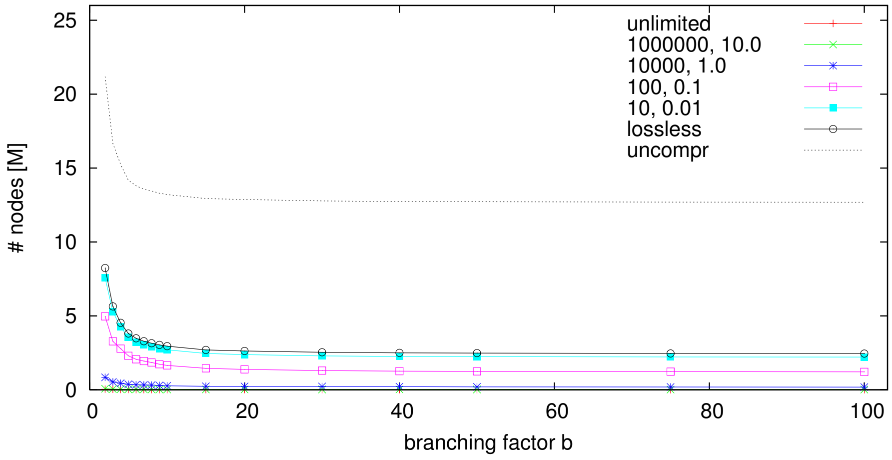


Figure 5.21: Absolute node count depending on maximum branching factor b.

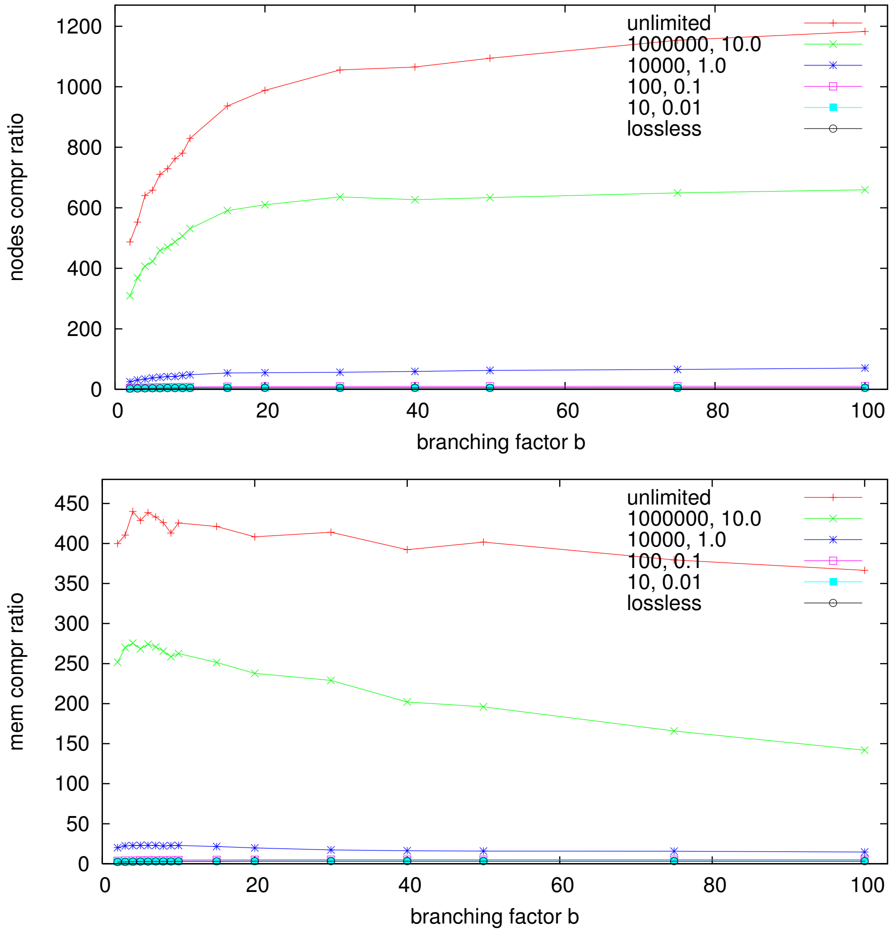


Figure 5.22: Compression ratios R_{nodes} (top) and R_{memory} (bottom) vs. maximum branching factor b.

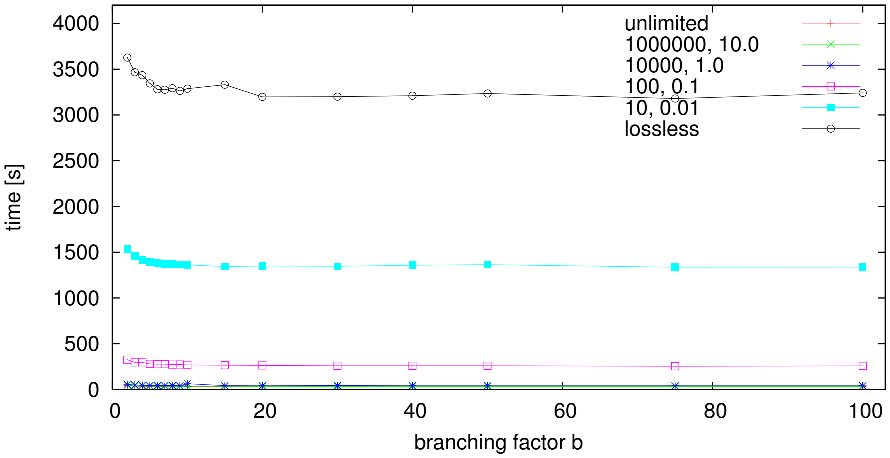


Figure 5.23: Compression run-time vs. maximum branching factor b.

Corollary 3. *For related traces that are identical with respect to the following properties:*

- *compression parameters (deviation bounds abs, rel, etc. and branching factor b, etc.),*
- *application code (with deterministic execution),*
- *application specific parameters (algorithm, domain decomposition, ...), and*
- *platform (architecture, CPU and memory speed, communication and I/O speed)*

the compression ratios R_{nodes} and R_{memory} are usually similar. The following properties can be changed without significant influence on them:

- *number of processes/threads and*
- *different problem sizes.*

The second experiment observes progressing compression for a single parallel trace. The data points show the compression ratios when processing the trace from the beginning to a certain point. Figure 5.25 reveals the progressing compression behavior for a single trace with 61 M records for varying compression parameters (abs, rel). The rightmost data points correspond to the rightmost ones in Figure 5.24.

Particularly, the large scale compression example shows an obvious change in behavior at approximately $x = 25\,M$ records. The same effect can be found in the other examples at the same point. Before this critical point, there is a monotone increase in R_{nodes}, followed by a slight decline followed by a further gradual ascent. The end point of the decline phases is *not* identical over all examples. Instead, it ends sooner for large scale compression and later for small scale compression.

The reason for the three phases can be found in corresponding phases of the execution of the SMG2000 code. At approximately $3\,s$ run-time out of $50\,s$ total run-time[2], the application switches from initialization to the solver phase. At first, the CCG compression adapts to reappearing patterns of the initialization phase. As soon as the solver phase starts, new patterns have to be adapted. This requires many new graph nodes, which decreases the compression ratio. Dissmissing supply of representing nodes at this point would have minimal influence to the over-all compression result. As soon as the new patterns are absorbed and repeated more or less regularly, almost no new nodes are added. Thus, compression ratios start to increase again. With small scale compression (many representing nodes) this takes longer as with large scale compression (fewer representing nodes).

For all experiments shown here, the behavior of $R_{memory}$is very similar to approximately $\frac{1}{2} \cdot R_{nodes}$.

---

[2] Note, that record count is not proportional to run-time even though both are monotonous.

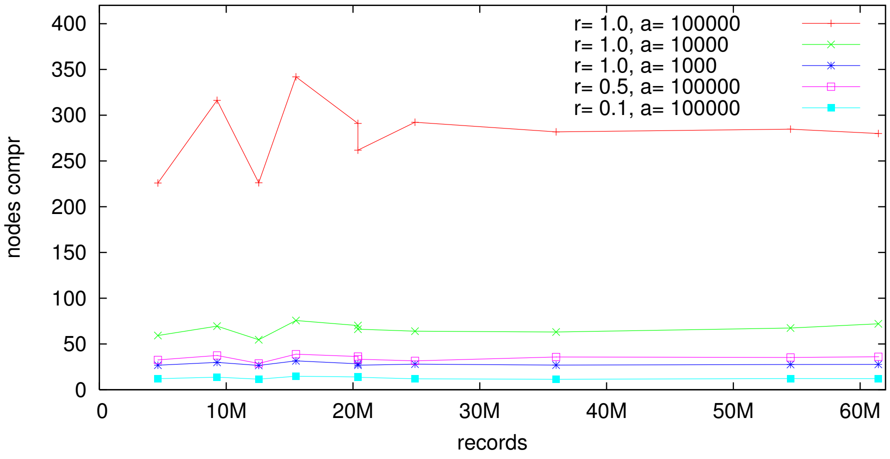


Figure 5.24: Influence of trace size (record count) to node compression ratio R_{nodes}. A series of different traces with varying record count (x-axis) show corresponding compression ratios R_{nodes} with different deviation bounds (color).

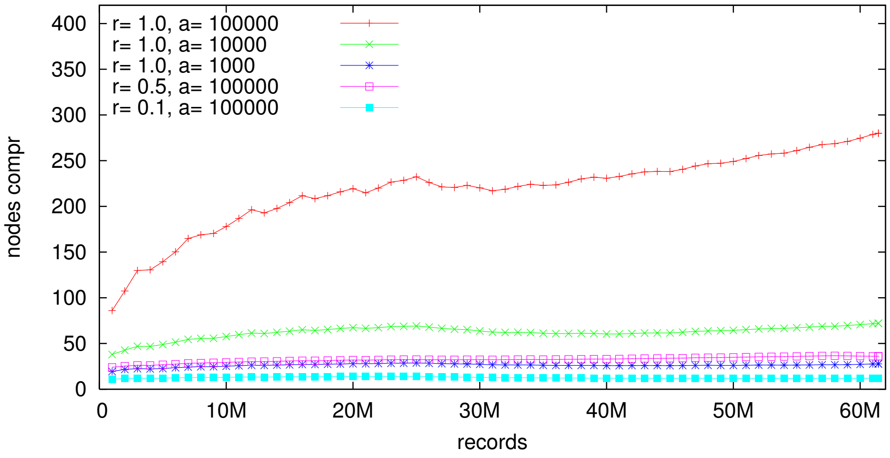


Figure 5.25: Progressing compression ratio R_{nodes} during the compression of a trace with advancing record count (x-axis) and different deviation bounds (color).

Corollary 4. *The compression ratios R_{nodes} and R_{memory} are neither constant nor monotonous during the course of compression. Instead, there may be phases with different compression ability that affect the over-all compression proportionately.*

5.2.5 Influence of Search Length Parameter

The effort for CCG compression is largely determined by search for replacement nodes. In particular, linearly traversing lists of candidate nodes dominates the over-all effort. Section 4.2.2 presents an algorithm for searching replacement nodes that contains a search length parameter L. This parameter effectively reduces the search effort and the over-all compression run-time. The following experiments investigate its influence on the compression ratio. Compression of the example trace is performed with varying L from very small to large values and infinity. This is done with a range of deviation bounds from lossless compression over small, medium and large scale compression.

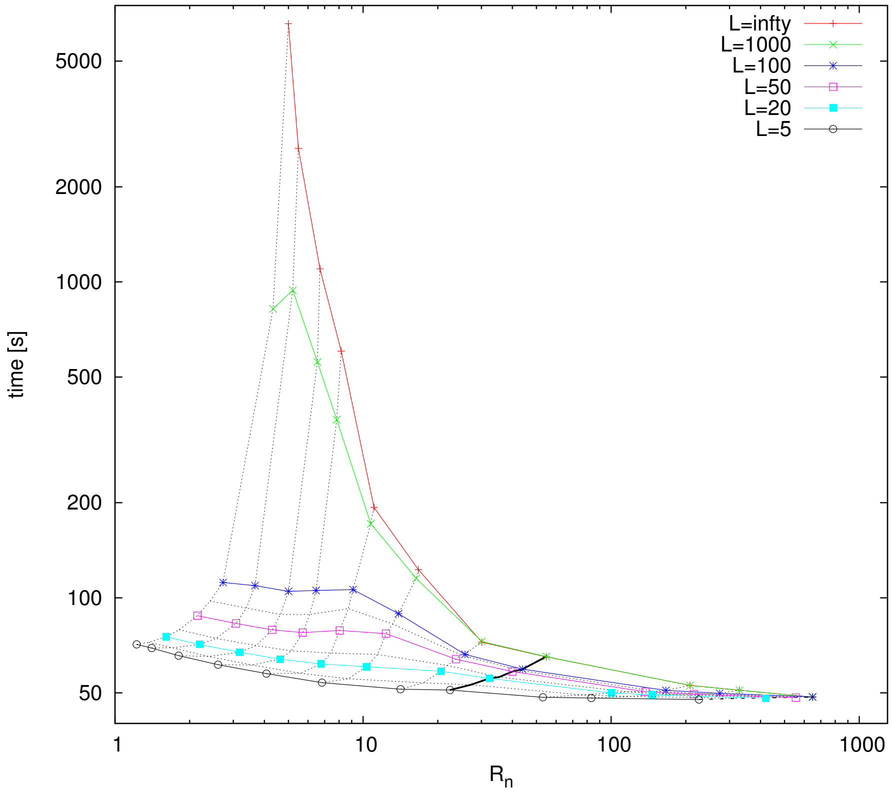


Figure 5.26: Influence of search length parameter L to the ratio R_{nodes} and run-time t. The dotted gray lines connect samples with identical deviation bounds abs and rel from lossless compression (leftmost) to large scale compression (right). Note the double logarithmic scales.

Figure 5.26 shows the node compression ratio and the run-time for different values of L and various deviation bounds $abs = 0 \ldots 1\,000\,000$ and $rel = 0.0 \ldots 10.0$. Both, R_{nodes} and t decrease with smaller L. For example with medium scale compression ($abs = 10000$, $rel = 1.0$) R_{nodes} is 55 for unlimited search length L and decreases to 44 for $L = 100$ while compression run-time is slightly reduced from $65\,s$ to $59\,s$. For smaller deviation bounds, the benefit of limited search length is higher, lossless compression with a reduction from $L = \infty$ to $L = 100$ to $L = 5$ shows a dramatic decrease of R_{nodes} from 5.0 to 2.7 to 1.22 while the corresponding compression time falls from $6566\,s$ to $111\,s$ to $71\,s$! With large deviation bounds ($abs = 100,000$, $rel = 5.0$) the compression ratio falls from 329 to 83 while L is reduced from 1000 to 5. Still, the run-time decreases only slightly from $51\,s$ to $48.2\,s$.

Therefore, the limitation of search length is an important feature for small scale compression, but less relevant for larger deviation bounds. A rather large default value for L will allow almost optimal results for large scale compression while effectively reducing the computational effort for small scale compression.

Corollary 5. *For small scale compression, limited node search length L allows a notable reduction in compression effort (run-time) with only small decrease in compression ratios R_{nodes} and R_{memory}. For large scale compression, there is almost no influence with sufficiently large L. Very small L notably inhibit compression with only minor reduction of run-time.*

5.3 Advanced Construction and Compression Algorithms

In addition to the standard CCG construction and compression algorithms investigated before, the following experiments evaluate advanced construction and compression procedures. At first, it covers *re-compression* of already compressed CCGs. Then, adaptive compression is demonstrated which determines actual deviation bounds automatically in order to comply with given resource limits. Finally, serialization and restore operations are presented.

5.3.1 Re-Compression of CCGs

CCG re-compression introduced in Section 4.4.1 allows to reduce an existing CCG by increasing the deviation bounds for soft properties. This makes graph nodes compatible with respect to the new deviation bounds that were not with respect to the previous ones. Figure 5.27 shows the example compression ratios for the original compression $R_{nodes}^{original}$ with $(abs = 100, rel = 0.1)$ compared with the re-compression R_{nodes}^{re} with $(abs = 1000, rel = 0.5)$ as well as renewed compression R_{nodes}^{new} with $(abs = 1000, rel = 0.5)$. As expected, the re-compression results lie between the original and the renewed ones:

$$R_{nodes}^{original} \leq R_{nodes}^{re} \leq R_{nodes}^{new} \qquad \text{with} \qquad R_{nodes}^{original} : R_{nodes}^{re} : R_{nodes}^{new} \approx \text{constant}. \tag{5.5}$$

Figure 5.28 shows the corresponding run-time behavior for all three cases. Obviously, the re-compression is much slower than either the renewed compression or the original compression. There is a distinct influence of b to the run-time for re-compression. This reflects the theoretical effort for re-compression $O(d \cdot n \cdot (l + b))$, where smaller b cause larger d and larger n, compare Section 4.4.1, Equation (4.24). For small $b \leq 15$ this causes a substantial disadvantage of several factors.

Due to the reduced compression ratios achieved and the memory fragmentation problems (compare Section 4.4.1), the re-compression algorithm seems not desirable as a stand-alone operation. Yet, it may be useful as part of other algorithms, for example merging of independent CCGs, see Section 4.4.2.

Corollary 6. *CCG re-compression with relaxed deviation bounds is much slower than compression with the new deviation bounds and achieves limited compression ratios.*

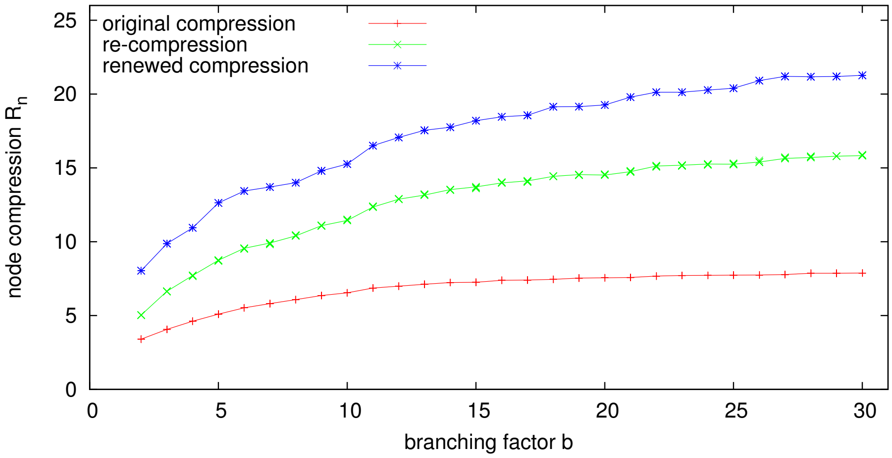


Figure 5.27: Comparison of node compression ratios R_{nodes} for original compression, re-compression and renewed compression with varying branching factor b.

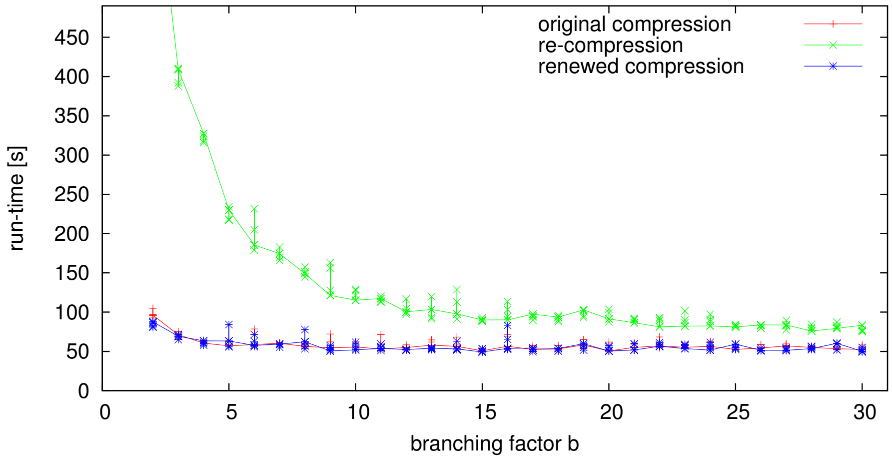


Figure 5.28: Compression run-time for original compression, re-compression and renewed compression.

5.3.2 Adaptive CCG Compression

Adaptive compression primarily works like standard compression algorithm. In regular intervals it checks for compliance with ressource limits. If this turns out negative, a re-start is performed. This includes dismissing the current intermediate graph and relaxing deviation bounds, compare Section 4.4.3. For the experiments shown here, exponential relaxation strategies with constant factors $f = 1.5 - 8$ are used. Starting with initial deviation bounds r_0 and a_0 the bounds are $a_i = a_0 \cdot f^i$ and $r_i = r_0 \cdot f^i$ after the i'th relaxation step. The experiments shown in Figures 5.29 and 5.30 use initial parameters $r_0 = 0.01$ and $a_0 = 10$. For the example trace with 24.9 million events the node count resource limits were $n_{max} = 100\,000 - 3\,000\,000$, which are unachievable with initial deviation bounds.

Figure 5.29 shows that the node compression ratio R_{nodes} grows exponentially with the number of re-starts, i.e. it grows proportionally to the deviation parameters. For the larger resource limit ($n_{max} = 3\,000\,000$) only one or two re-starts are necessary, for the smallest resource limit ($n_{max} = 100\,000$) already 4 to 21 re-starts are needed. This results in very large final deviation bounds. As expected, for larger f fewer re-start steps are observed. Yet, this may lead to exaggerated final deviation bounds, which are much higher than the minimal deviation bounds that would comply with the resource restrictions.

Figure 5.30 shows the run-time behavior for the same experiments. As anticipated, the run-time decreases with larger f, i.e. fewer re-start steps before reaching the same final deviation parameters. Yet, with constant f and decreasing resource limits the total run-time is not necessarily increasing even though the number of re-starts grows. Instead, run-time is actually reduced in most of the cases. The reason for this at first surprising behavior is that smaller ressource limits make the compression algorithm spend less time with very small deviation bounds, because re-start occurs earlier, compare Section 5.2.1.

Corollary 7. *Adaptive compression is able to quickly determine suitable deviation bounds for given resource limits. Faster relaxation strategies lead to smaller total run-time but allow less fine-grained control over final deviation bounds.*

In above experiments, the decision about a re-start is made after the resource limits are exceeded. In fact, it would be sufficient to predict that it will be exceeded with current deviation bounds. Simple heuristics might be designed, for example: *restart if less than x percent of records account for ressource consumption of more than x percent of the limit.* Yet, compression behaves non-uniformly with cumulating record count, compare Section 5.2.4 and Figure 5.25. Therefore, such heuristic approaches would reduce the over-all effort for the cost of even further exaggeration of final deviation bounds.

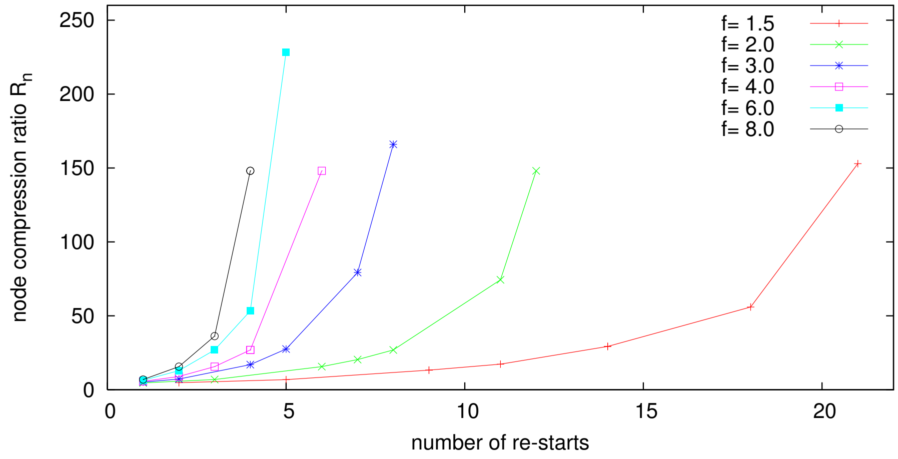


Figure 5.29: Compression ratio R_{nodes} vs. the number of re-start steps for adaptive compression with limited node count $(100\,000, 250\,000, 500\,000, 750\,000, 1\,000\,000, 2\,000\,000, 3\,000\,000)$.

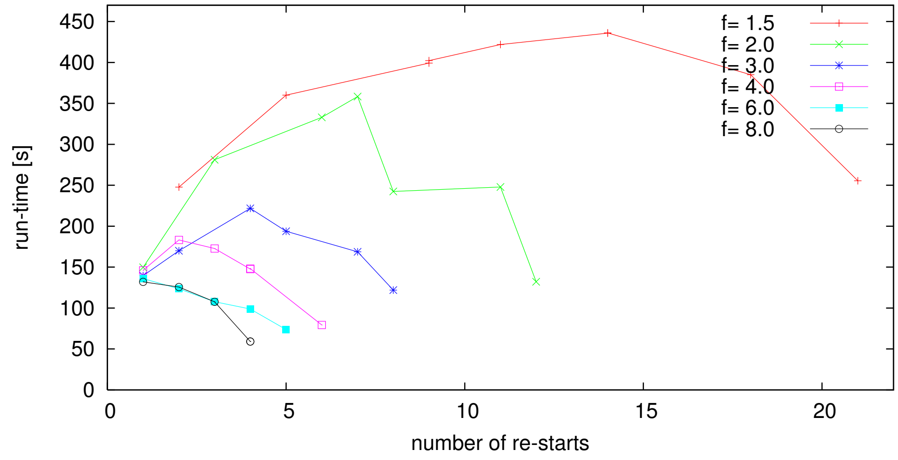


Figure 5.30: Adaptive compression run-time including multiple re-start steps vs. the number of re-starts.

5.3.3 Serialization and Restore of CCGs

CCG serialization and restoring is evaluated with the original *SMG2000* trace which uses 5.5 MB to 463 MB memory with large scale or small scale compression. The experiments support the predictions of linear effort for both operations, compare Section 4.6. Both operation are very fast compared to repeated compression. Figure 5.31 shows almost perfect linear behavior for serialization and restore run-time with respect to memory consumption m. The corresponding sustained speed ranges from 25 to 29 MB/s. The speed of the restore operation is slower due to the necessary pointer translation. It shows equally linear behavior with over-all speed from 8 to 9.5 MB/s.

The speed with respect to node count (not shown) ranges from 90 000 to 255 000 nodes/s for serialization and from 30 000 to 75 000 nodes/s for restoring. It is only roughly constant and has a distinct influence of b especially for smaller n. This suggests that the speed is actually bound by the data volume. The effects on speed with respect to node count are induced by the proportions between m, b and n.

Corollary 8. *Serialization and restore of CCGs show strictly linear effort with respect to the data volume. Both are very fast in comparison to repeated CCG construction and compression.*

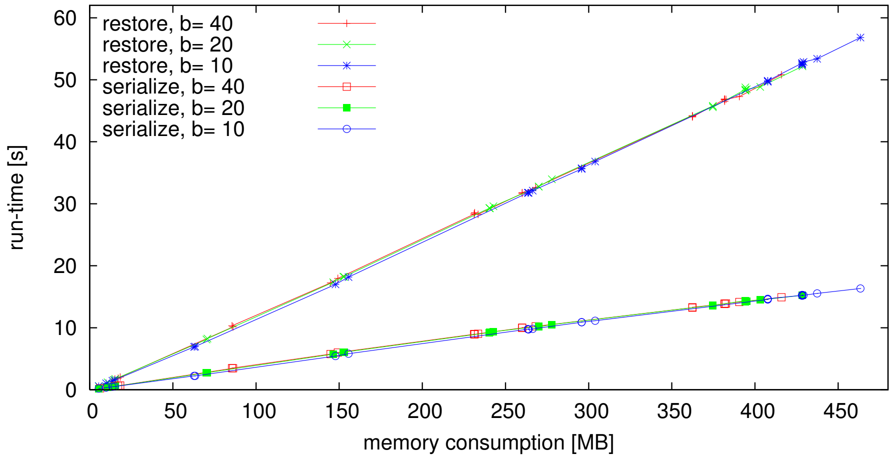


Figure 5.31: Run-Time for CCG serialization and restore vs. CCG memory consumption m.

5.4 Cached Summary Queries

The very purpose of CCGs is the analysis of application traces. CCG construction and compression can be done via batch processing early. Restoring a CCG from serialized form is fast, as shown above. Only querying is always to be performed on demand. Usually, many successive queries are required to generate the insight to the human user. Therefore, query performance is important in order to allow a convenient workflow which is not impaired by long waiting periods.

Statistic summary queries are the most frequent and most performance demanding queries on CCGs. Many important evaluation procedures rely on multiple statistic queries, e.g. complex information like the well known timeline diagram [BNS00, BHNW01] are derived from multiple summary queries. Furthermore, it includes the time search query when restricting the time interval in question.

Most of the time, such kinds of queries are utilized successively for overlapping areas of a trace. Most frequently this happens for nested time intervals as the user usually approaches traces from a global overview down to specific interesting details which are revealed only on this path. This can be exploited for notable performance improvement.

5.4.1 Cache Strategies

The *Cached Summary Queries* re-uses intermediate results of recent queries. This is beneficial when parts of a CCG are to be evaluated multiple times in successive queries. And it is also advantageous for an isolated query, because some sub-graphs of a CCG are traversed multiple times due to compression.

In this section, summary queries are examined following the path from global overview to a small detail. On all time intervals, performance is shown for initial queries as well as for successive queries taking advantage of cached intermediate results of former queries. Note that in a real world scenario only the very first global query would be a initial one while all following queries take full advantage of caching.

In order to balance the cache memory requirements and the performance improvement of the queries a partial caching scheme is applied, i.e. only selected nodes are subject to caching (respectively the intermediated results associated with that nodes). Here, all nodes with a certain depth level are cached as well as all nodes referenced more often than a given threshold.

The according caching parameters are like follows. The *cache modulus* parameter $c_m = 10$ triggers caching for all nodes with

$$depth(node)\ mod\ c_m = 0. \tag{5.6}$$

Hence, after at most c_m steps of depth traversal there will be a node with cached intermediate results. Furthermore, caching is performed for all nodes that are referenced more often than a certain threshold $c_r = 5$, the *cache references* parameter. Section 4.5.3 provides theoretical model of query speed.

5.4.2 Experiment Results

The following experiments cover two scenarios. At first, initial queries with previously empty caches. Usually, those are issued only once in a series of interactive queries. Furthermore, successive queries with pre-filled caches. Both experiments cover queries on CCGs with varying total sizes from $n = 27\,000$ to $n = 4\,200\,00$ nodes[3]. For every example, different sub-intervals of the complete time range have been selected, including the full range from $51\,965\,000$ to $31\,005\,400\,000$ ticks (approx. $17ms$ to $10.3s$) and intermediate time intervals of length $t = 10^{10}$ ticks down to $t = 10^4$ ticks (approx. $3.3s$ to $3\mu s$). The time intervals corresponds to irregular sub-sets of the CCG's nodes. Their size can be assumed roughly proportional to the uncompressed and compressed node counts N and $n \leq N$ in the interval.

Figure 5.32(top) shows that the initial query time is roughly proportional to the length of the query intervals t (roughly constant distance with respect to logarithmic time axis). The influence of the total node count n (x-axis) is almost linear, see also non-logarithmic in Figure 5.33(top).

Successive queries show a logarithmic behavior with respect to the node count n, see Figure 5.32(bottom) and Figure 5.33(bottom) with non-logarithmic time axis. Compared to initial queries with large t there is a substantial performance gain of more than one order of magnitude. For smaller queries there is only a moderate performance advantage. Yet, such queries are quick, already. For small $t \leq 10^4$ the results for both cases are almost constant below 0.5 ms.

Figure 5.33 gives a detailed look at the experiment with large query interval of lengths t with non-logarithmic time axis. In particular, it shows the influence of the branching factor $b = 10$, 20, 40 on query performance. For initial queries this results in a nearly linear behavior with respect to n with slightly slower speed for larger $b = 40$ – see Figure 5.33(top). For successive queries the influence of n is not linear any more but logarithmic! Furthermore, there is a a distinct advantage for smaller $b = 10$ – see Figure 5.33(bottom).

Corollary 9. *The Cached Summary Query achieves different speed for initial queries with previously empty caches and for successive queries with pre-filled caches. The computational effort of initial queries is proportional to the compressed number of nodes n_{part} covered by the query time interval $O(n_{part})$. Successive queries are much faster than their initial counterparts by more than an order of magnitude. Their computational effort is reduced to $O(\log n_{part})$.*

5.5 MPI Send-Receive Matching

The matching of MPI send and receive events is based on event iterators, compare Section 3.4.2). The matching algorithm performs a partial de-compression by traversing all original events in correct order, see Section 3.4.5, and temporarily stores a sub-set of un-compressed events. Therefore, ressource usage could exceed the size of the underlying CCG. This halfway contradicts the idea of evaluation with a compressed data structure. However, iterating over the events of all processes in an alternating scheme allows to limit the amount of pending events at any point in time, as presented in Section 4.5.5.

---

[3] The CCG size may vary due to different compression or due to different trace sizes

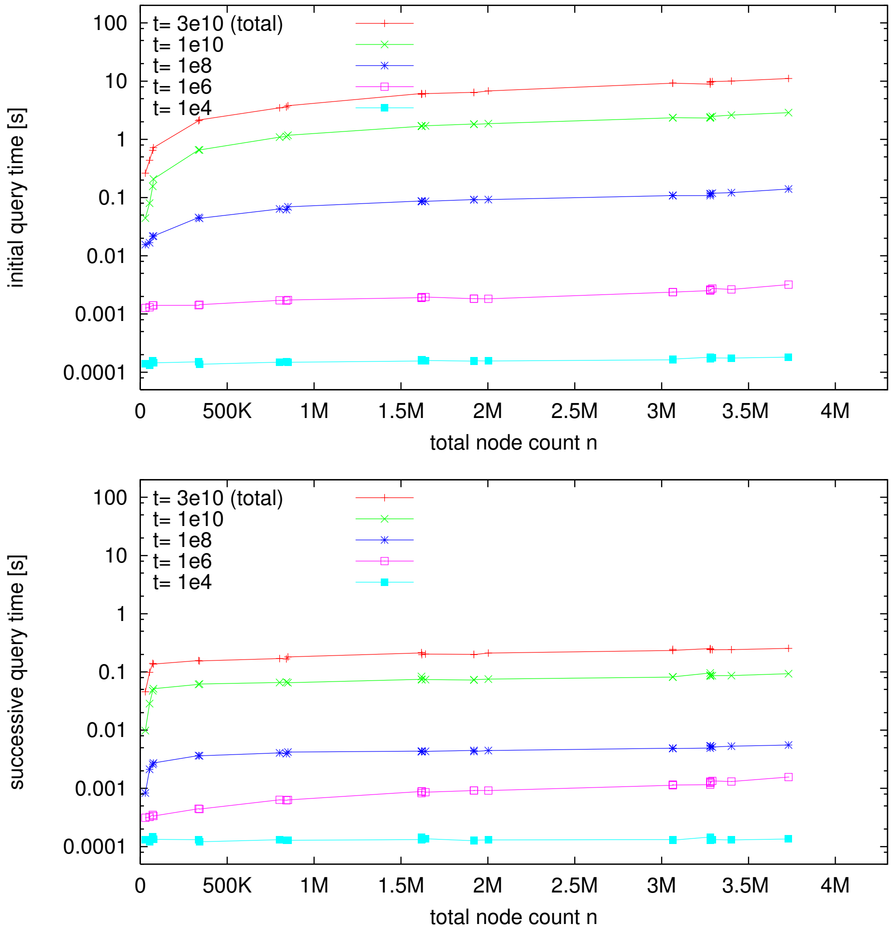


Figure 5.32: Comparison of query run-time for initial (top) and successive (bottom) queries with respect to the total node count n with fixed branching factor $b = 20$. The query intervals vary from length $t = 1 \cdot 10^4$ to $3 \cdot 10^{10}$. (Mind the logarithmic time axis.)

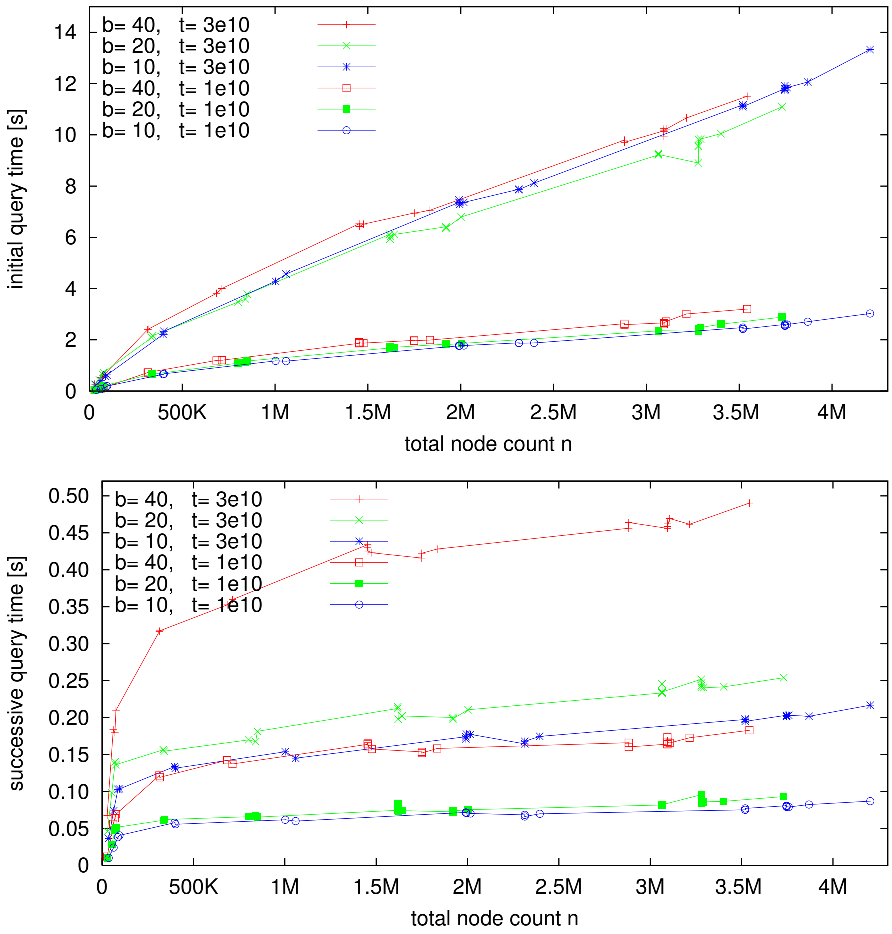


Figure 5.33: Initial (top) and successive (bottom) query run-time with respect to the total node count n and varying branching factor $b = 10,\ 20,\ 40$ with linear time axis.

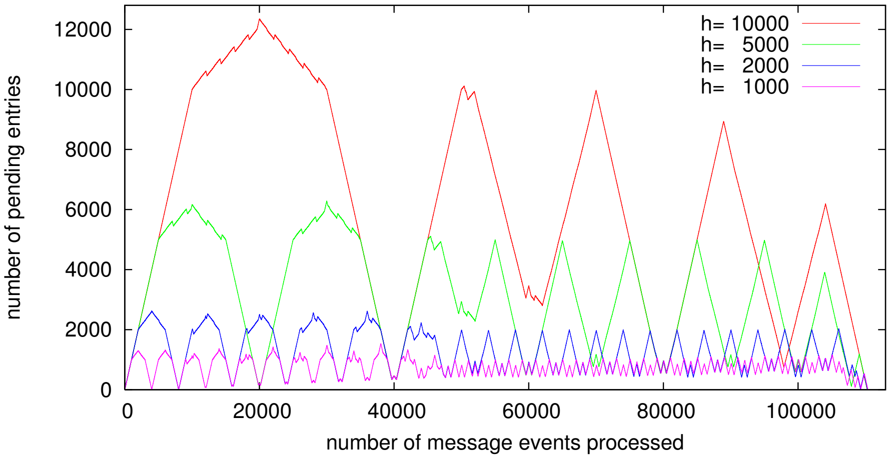


Figure 5.34: Temporary ressource usage of the matching algorithm for MPI send and receive events with 4 processes and varying phase length h. The respective maxima are proportional to h.

Figure 5.34 illustrates the number of pending events with a small example trace with $p = 4$ parallel processes and $s \approx 110\,000$ evenly distributed send and receive events. After every h message events the operation switches to the next process trace. In the beginning there are regular phases of ascending and descending resource usage with length h. With progressing operation and with growing number of traces processes the phases become less distinct.

The maximum resource usage is obviously proportional to h and the minimum is almost 0 which is reached regularly after $p = 4$ phases. The behavior in Figure 5.34 represents a reasonable real-world scenario. With additional pre-conditions for the distribution patterns of send and receive events upper bounds for the resource usage can be deduced. The (improbable) worst case scenario still requires $s/2$ temporary entries for s message events.

The number of message events s is fixed for a given trace and is not influenced by the compressed node count n, the ratios R_{nodes} and R_{memory} or the branching factor b. Accordingly, the speed of the matching operation is almost constant at $\approx 5\,500$ events/s. The speed is almost unaffected by the phase length h. This is consistend with the $O(1)$ access to the list of pending events, compare Section 4.5.5.

5.6 Recommended Parameter Settings

From all the experiment results shown before a recommendation for CCG parameter settings can be derived. On one hand, there are deviation parameters for soft properties. On the other hand, there are internal parameters of the CCG compression and evaluation algorithms. Below, reasonable defaults are given for both if no specific settings are known.

5.6.1 Deviation Bounds for Soft Properties

The most important compression parameters are the deviation bounds for soft properties. In all previous experiments the deviation bounds abs and rel for time durations were presented. This are good examples for all remaining deviation bounds because all enforce either an *absolute* limit similar to abs or a *relative* restriction proportional to the particular value similar to rel. The following list contains all deviation parameters supported in the current implementation:

- absolute time deviation for time stamps (abs),
- relative time deviation for time durations (rel),
- relative deviation of counter values that relate to time durations ($r_{counter}$),
- absolute deviation of bytes in point-to-point send or receive operations ($len_{send/receive}$),
- absolute deviation of time spent in send or receive operations ($dur_{send/receive}$),
- absolute deviation of bytes exchanged in collective communication ($len_{collective}$), and
- absolute deviation of time spent in collective communication operations ($dur_{collective}$).

The setting of deviation parameter will affect performance as well as the accuracy with respect to soft properties. Both goals are conflicting. Always, the particular field of application has to determine the accuracy necessary. Within this range, the deviation bounds should be specified as large as possible, because the most restrictive one would limit the over-all compression ability.

If in doubt, they should rather be more restrictive. However, even for very accurate evaluation small deviations are usually acceptable, at least in the same order of magnitude as the measurement error. Usually, all of the deviation bounds aim for a similar degree of accuracy. It seems unusual to have some very restrictive and very tolerant bounds at the same time. Yet, there is one exception: Deviation bounds to (currently) nonrelevant properties should be unlimited.

The following examples of real-world scenarios shall illustrate two typical settings.

Example Scenario: Visual Trace Analysis

For trace visualization and interactive analysis similar to Vampir the deviation bounds for all properties of interest should be rather restrictive in order to provide a realistic impression of the dynamic run-time behavior. Still, there is room for lossy compression as certain small timing fluctuations are irrelevant.

An absolute time deviation of $5\,\mu s$ and a relative time deviation of 5% could be typical conservative settings. This would basically limit small time durations to 5% deviation (even if it is less than $5\,\mu s$) and large durations to $\pm 5\,\mu s$ (even if it is less than 5%). With the particular timer resolution of for example 3 GHz ($1/3\,ns$ per tick) this translates to:

$$\begin{aligned} abs &= \frac{1\,\mu s}{\text{timer resolution}} = \frac{1\,\mu s}{1/3\,ns} = 3000 \text{ and} \qquad (5.7) \\ rel &= 5\% = 0.05. \qquad (5.8) \end{aligned}$$

Note, that the absolute time deviation should be below of half the minimum message latency to avoid reversed messages, i.e. messages that look like being send backwards in time. Therefore, above example would be valid for a minimum message latency of $10\,\mu s$ or above which is a realistic value.

Example Scenario: MPI Communication Re-Play

For MPI message replay, which is for example used for benchmarking, the deviation parameter settings would look completely different. As the original run-time behavior is irrelevant, the regarding deviation bounds would be unbounded.

On the other hand, all communication related deviation bounds like $len_{send/receive}$, $dur_{send/receive}$, $len_{collective}$, and $dur_{collective}$ should be set to zero or very restrictive. However, if certain properties always have zero deviation bounds in a particular field of application, they should be treated as hard properties instead of soft properties. This would mean no additional restriction to compression but would notably improve compression speed, compare Section 3.3.3.

5.6.2 Algorithm Parameters

The search length parameter l and the maximum branching factor b have no influence on data accuracy but only on compression capability and performance of compression and evaluation.

The search length parameter l should always be bounded according to the results presented in Section 5.2.5. A rather large value like 1000 will effectively avoid excessive compression run-time for small scale compression. At the same time, it will hardly influence the maximum compression ratios achievable with large scale compression.

The maximum branching factor b affects CCG compression as well as querying. In general, compression performance and compression ratios improve with larger b. For Cached Recursive Queries a rather small value of b provides an advantage in terms of query speed. For larger b querying becomes slower even though there are less nodes in the first place, because large b improve compression. As both effects are opposed, b can be set in favor of compression or for the benefit of query speed. As a good compromise a setting of $b = 10 \dots 30$ is recommended. This will achieve good query performance while the penalty for compression with small $b < 10$ is avoided.

5.7 Comparison of the CCG implementation to State-of-the-Art Trace Analysis Tools

The preceding results showed that the CCG approach is clearly suitable to build interactive analysis tools on top of it. It is able to handle very large traces effectively and efficiently. Now, the last part of this chapter presents a comparison to the state-of-the-art, in particular the Vampir and VampirServer tools, compare Section 2.2.1. For the two commercial tools as well as for the CCG implementation, the two most important features are compared, which are the memory consumption, the loading time from files and the speed of statistic summary queries. Again, the SMG2000 example trace is used, see Section 5.2.

The total memory consumption for all cases is given in Figure 5.35. Vampir uses quite substantial amounts of memory, probably due to inefficient re-allocation of memory. It's successor VampirServer shows a memory consumption in the range of the uncompressed trace file size, as expected. The CCG implementation requires notably less memory from 430 MB to 100 MB to 10 MB for minimum, medium scale and maximum compression. This underlines the main advantage of the compressed CCG approach.

The following comparison looks at loading time from trace files, see Figure 5.36. Vampir and VampirServer are both faster than the CCG construction, because they contain no compression phase. VampirServer shows an notable advantage over the sequential Vampir version, in particular with more distributed worker nodes. Loading from multiple parallel trace files (yellow) is even faster than loading from a single trace file (red). CCG construction and compression is substantially slower for minimum and small scale compression, but shows similar speed with medium scale compression and beyond.

As an alternative during an interactive workflow, the CCG may be restored from a previously compressed and serialized version. The restore operation is much faster than CCG creation. With medium scale compression and below, it is only slightly slower than loading in Vampir or VampirServer. With large scale compression and above it is much faster however, due to the very small data volume.

The last comparison looks at the run-time for statistical queries, compare also Section 5.4. The run-time of three queries covering the full trace as well as parts of $1/3$ and $1/30$ of the complete time interval is shown in Figure 5.37. The event count is approximately proportional to the interval length.

The CCG-based cached query works with pre-filled caches (successive query) because this represents the typical situation during interactive trace analysis, compare also Section 3.4.3. It is faster than the corresponding query in VampirServer with a single worker process. It is more than twice as fast with

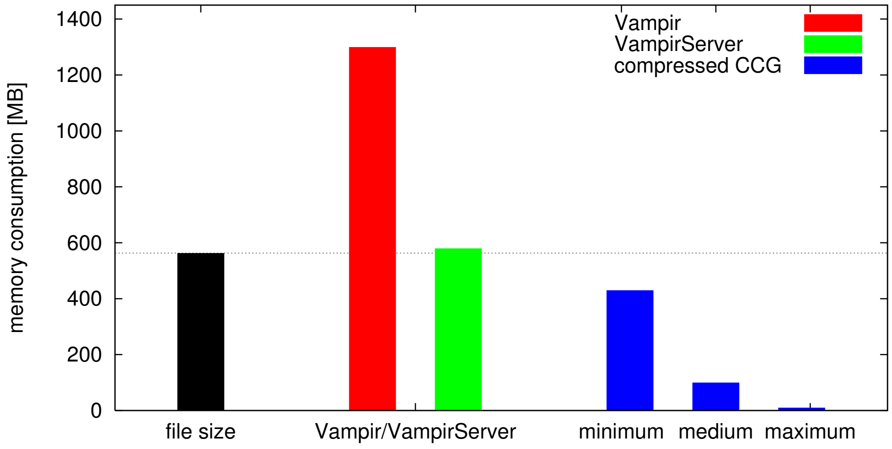


Figure 5.35: Comparison of the total memory usage of Vampir, VampirServer, and the compressed CCG approach with different compression settings.

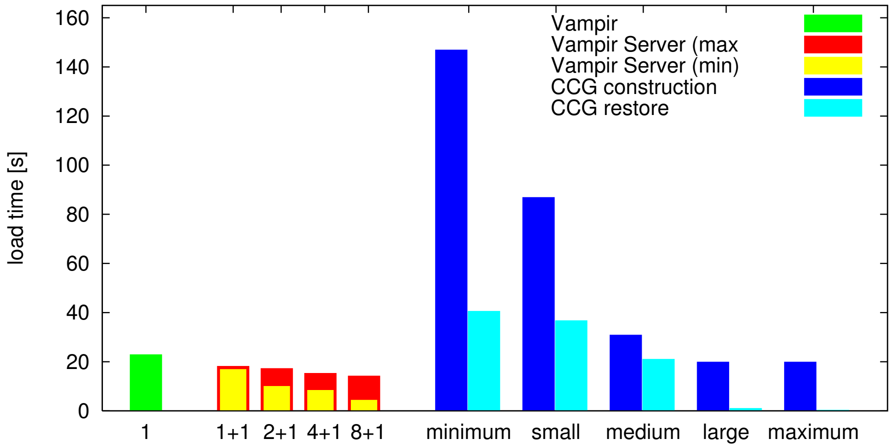


Figure 5.36: Comparison of the loading time for Vampir (sequential), VampirServer (with 1 to 8 worker processes) and the CCG approach with different compression settings (sequential).

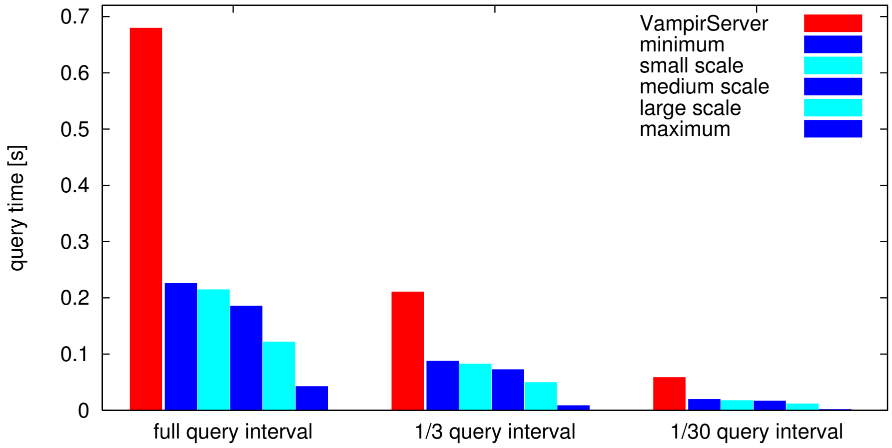


Figure 5.37: Comparison of the run time of cached summary queries for different query time intervals.

	Vampir	VS 1+1	VS 2+1	VS 4+1	VS 8+1
load trace	23s	17.0 - 18.3s	10.1 - 17.4s	8.5 - 15.4s	4.5 - 14.3s
sum query 1/1	7s	0.680s	0.352s	0.181s	0.104s
sum query 1/3	2s	0.211s	0.114s	0.063s	0.039s
sum query 1/30	< 1s	0.059s	0.037s	0.027s	0.021
sum query 1/300	< 1s	0.010s	0.010s	0.009	0.011

Table 5.1: Loading and querying times of the default example trace with Vampir and VampirServer.

	min. compr.	small scale	medium scale	large scale	max. compr.
load + compress	147s	32 - 87	20 - 31s	20s	20s
restore	40.675	36.852	21.142	1.128	0.422
initial					
sum query 1/1	9.983	8.621	5.543	0.663	0.255
sum query 1/3	2.884	2.379	1.679	0.156	0.044
sum query 1/30	0.742	0.602	0.440	0.055	0.028
sum query 1/300	0.141	0.116	0.086	0.022	0.016
successive					
sum query 1/1	0.226	0.215	0.186	0.122	0.043
sum query 1/3	0.088	0.083	0.073	0.050	0.009
sum query 1/30	0.020	0.018	0.017	0.012	0.002
sum query 1/300	0.005	0.005	0.004	0.003	0.001

Table 5.2: Loading and querying times of the default example trace with the CCG implementation.

minimum and small scale compression and up to ten times as fast with maximum compression. The query run-time values for VampirServer and the CCG implementation cover only the queries itself excluding any output or screen rendering operations. Note that for VampirServer there is an additional overhead because of network communication with the master process (compare [BNM03, Bru08]). However, this overhead should be constant with a single worker node. Therefore, it is assumed smaller than the minimum query run-time of 0.01 s.

The complete list of all run-time results can be found in Table 5.1 for Vampir and VampirServer and in Table 5.2 for the CCG implementation. The tables reveal that VampirServer already provides a notable advantage over the sequential Vampir version, even with only a single worker node. Furthermore, they show that VampirServer is able to achieve quicker query responses with more worker processes. The scaling is almost optimal. However, VampirServer cannot scale beyond n worker nodes for a trace with n processes ($n = 8$ in this case). With 4 or 8 worker nodes on 4 or 8 CPUs VampirServer is approximately as fast as the CCG implementation with medium or large scale compression on a single CPU.

Of course, the presented comparison to a state-of-the-art software tool is not ultimate. In particular, it does not investigate further ways of optimization and tuning that may be possible for either tools. Yet, the comparison shows that the CCG implementation is able to achieve comparable performance as the product quality software tool VampirServer while providing substantial resource reduction. Alternatively, it can provide enhanced scalability with the same resources. Furthermore, it allows a new mode for event trace processing: Coarse-level trace analysis with increased deviation bounds may be a desired compromise solution when the accurate procedure becomes impossible due to exorbitant data volumes. And last but not least, the CCG approach may be combined with distributed trace analysis, as introduced in Section 4.7, in order to achieve unmatched scalability.

6 Conclusion and Outlook

Finally, this chapter provides a conclusion of the thesis and gives an outlook on future work.

Summary and Conclusion

This thesis proposed a new data structure named Complete Call Graph for the representation of event traces in main memory. It is especially designed according to the requirements and conditions of automatic and interactive performance analysis tools.

After introducing the principles of the new event trace representation and the method for domain specific data compression, a comprehensive discussion of all essential algorithms for construction, compression and analysis of the CCG data structure confirmed that it is an adequate alternative to the established data representation methods using sequential arrays or lists.

An elaborate theoretical and experimental evaluation with real-world examples demonstrated the data compression ability according to the reduction of memory consumption and the reduction of the number of graph nodes. The former is measured by the memory compression ratio R_{memory} and achieved results from ≈ 3 for lossless or small scale compression, to > 10 for medium scale compression, and to more than 100 for large scale compression. The latter is measured by the node compression ratio R_{nodes} and ranged from ≈ 5 for lossless or small scale compression, to > 40 for medium scale compression, and up to more than 500 for large scale compression. Experiments with synthetic examples indicated that the maximum compression can grow infinitely for very regular traces.

The newly designed analysis algorithms based on the CCG data structure were shown to profit twofold, from the reduced memory footprint as well as from the decreased computational effort. The degree of compression can be controlled by the deviation bounds for soft properties which allows to bias the approach either towards high accuracy with moderate compression or towards extreme compression with limited accuracy.

Future Work

The future work based on the results of this thesis should to focus on three subjects: incorporating an implementation of the Complete Call Graph data structure into existing event trace analysis tools, incorporating new trends in trace-based performance analysis, and extending the notion of a compressed representation one step further to the graphical user interface.

The existing research implementation of the CCG data structure and the associated algorithms need to be developed into a robust, product quality implementation, in order to make it attractive for established trace analysis tools. This may include minor refinements for the benefit of the practical usage. The general feasibility has already been demonstrated by a prototype integration into the VampirServer tool [KBN05], see also Section 2.2.1 and Figure 2.1.

Furthermore, new trends in performance analysis need to be incorporated into the event trace analysis and visualization. This may include new event types, new performance metrics, and new objectives for optimization. One example is the trend towards energy efficient computing which requires to consider not only execution time but also energy consumption as the subject of optimization.

Future Research

The future research starting from the existing Complete Call Graph approach should go one logical step further, bringing the notion of *"information compression"* to the end-user.

This thesis has been demonstrating, how the CCG data structure can provide all building blocks of the classical paradigms of event trace analysis and visualization while notably reducing the memory consumption and the computational effort. This allows better performance of trace analysis tools, faster interactive responce and higher scalability for traces of growing size.

In addition, the CCG approach provides the potential to extend the scope of visualization and analysis tools by an automatic classification of regular and irregular behavior of repeated event sequences. In a first step, this would require to identify classes of related call sequences. The classes should contain all pairwise identical call sequences (sub-trees) irrespective of the run-time behavior and other soft properties. Furthermore, also structurally similar call sequences should be included, for example variable iteration counts within a otherwise identical sequence. Again, this will require an appropriate definition of *similar*. The CCG approach already provides a preliminary support for this classification. The S-lists in the management data structure for replacement nodes contain groups of related nodes (sub-trees) that differ only with respect soft-properties, see Section 4.2.1.

In the second step, the classes of related call sequences could be investigated for their dynamic behavior in order to automatically distinguish the frequent regular behavior from rare fluctuations and outliers. This rating could be presented to the user by a sophisticated visualization design, allowing a quick survey of regularity for large traces. Furthermore, the user could reliably judge the average regular behavior of essential program parts by investigating a single instance thoroughly. The irregular and maybe performance critical parts would still be available for separate consideration and comparison.

Similar to the CCG data compression, the *"information compression"* would allow the user to focus on the important information by ignoring redundancy due to repetitions. A first attempt towards this concept has been reported in [Voi06].

A Acknowledgements

At the end of writing, I'd like to express my thanks to all who supported me in accomplishing this thesis.

First of all, I want to thank Prof. Dr. Wolfgang E. Nagel for the opportunity to achieve this thesis and for many valuable motivations and stimulations. Also, I would like to thank my reviewers Prof. Dr. Wolfgang Lehner and Prof. Dr. Dieter Kranzlmüller for their advise and feedback.

Furthermore, I would like thank all colleagues at the former ZHR and today's ZIH at TU Dresden and at the Institute for Scientific Computing at the Department of Mathematics for the pleasant working environment during the past six years.

In particular, many thanks go to Holger Brunst, Claudia Schmidt, Sabine Vollheim, Guido Juckeland, Michael Kluge, Robert Henschel, Heike Jagode, Matthias Jurenz, Ronny Brendel, Jens Doleschal, Jacqueline Papperitz, and Matthias Müller for their support, their encouraging and critic feedback and their friendship. Moreover, I want to thank Eleonora Flach for her reviews of the manuscripts as well as Ulf Markwardt for caring about the tea.

And finally, very special thanks to my family for their unreserved support during a long time, in particular my parents Gisela and Klaus, my daughter Helena and my partner Anita.

I'm very grateful for the funding of parts of the research for my dissertation by the graduate school 191 of Deutsche Forschungsgesellschaft (DFG) "Werkzeuge zum effektiven Einsatz paralleler und verteilter Rechnersysteme" at TU Dresden and by the HPC-EUROPA project (RII3-CT-2003-506079) with the support of the European Community Research Infrastructure Action under the program FP6 "Structuring the European Research Area".

List of Figures

2.1 Overview of VampirServer's distributed architecture . . . 14
2.2 Vampir global timeline display . . . 14
2.3 Vampir local timeline display . . . 15
2.4 Vampir summary chart . . . 15
2.5 Vampir calltree display . . . 16
2.6 Vampir communication statistics display . . . 16
2.7 Global Paraver display . . . 19
2.8 Paraver's performance counter display . . . 19
2.9 Hierarchy of behavior patterns of Expert 3.0 . . . 20
2.10 Examples behavior patters for point-to-point messages . . . 20
2.11 The CUBE display . . . 21
2.12 CUBE display including the topology view . . . 21
2.13 Jumpshot 4 timeline display . . . 23
2.14 Jumpshot 4 statistics display . . . 23
2.15 Dewiz display of alternative outcomes of a wildcard communication . . . 25
2.16 Indication of special message properties in DeWiz . . . 25
2.17 Coordinated parallel break point in DeWiz . . . 25
2.18 TAU ParaProf display . . . 27
2.19 TAU ParaProf's call graph display . . . 27
2.20 TAU ParaProf's three-dimensional profile view . . . 27
2.21 TAU ParaProf's two-level displays for phase-based profiles . . . 28
2.22 The OTF storage scheme . . . 32
2.23 Collaboration diagram of tracing tools and formats from the TAU documentation . . . 35
2.24 The event type hierarchy defined by the EARL library. . . . 37
2.25 Mapping of time stamps to pixel positions for timeline rendering . . . 43

3.1 Example of a simple repeated call sequence . . . 50
3.2 Example of a Complete Call Graph of a single process . . . 51
3.3 Example of splitting wide graph nodes. . . . 53
3.4 Comparison of an uncompressed and a compressed Complete Call Graphs . . . 54
3.5 Accumulation of absolute time stamp deviation over multiple nodes . . . 57
3.6 Schematic diagram of successive queries onto a CCG . . . 59
3.7 Screenshot of Vampir's timeline display. . . . 60
3.8 Timeline rendering with color blending . . . 61
3.9 Matching of MPI send and receive events . . . 62
3.10 Matching of collective MPI communication events . . . 62

4.1 Step-by-step demonstration of CCG construction. . . . 67
4.2 Example of on-the-fly splitting of wide nodes . . . 68
4.3 Balanced splitting and minimum-depth splitting for wide nodes . . . 69
4.4 OOP class hierarchy for node data structures and TMP extension . . . 70
4.5 Successive compression in a CCG . . . 72
4.6 Hash data structure for graph nodes . . . 73

4.7 Searching and caching in S-Lists . . . 73
4.8 Effect of node search order for compression . . . 74
4.9 Comparison of graph nodes for absolute time deviation . . . 76
4.10 CCG construction and compression algorithm (Part I) . . . 78
4.11 CCG construction and compression algorithm (Part II) . . . 79
4.12 CCG re-compression algorithm . . . 81
4.13 Compression algorithm with adaptive deviation bounds . . . 84
4.14 Navigation in four directions inside a CCG. . . . 86
4.15 Recursive algorithm for fast time-stamp search . . . 87
4.16 Vampir color coded timeline display . . . 88
4.17 Uncompressed CCG with multiple evaluation paths . . . 89
4.18 Uncompressed CCG with critical evaluation paths . . . 89
4.19 Compressed CCG with multiple evaluation paths . . . 89
4.20 Improved timeline rendering with color blending . . . 91
4.21 Matching algorithm for MPI send and receive events . . . 92
4.22 Algorithms for CCG serialization and restore . . . 93
4.23 Examples of connected and distributed CCGs . . . 95
4.24 Decomposition for distributed CCGs . . . 96

5.1 CCG compression model . . . 100
5.2 Example of non-monotonic compression . . . 101
5.3 Best-case CCG compression . . . 102
5.4 Best-case CCG compression with an unbalanced binary recursion pattern . . . 102
5.5 Worst-case CCG compression with random call patterns . . . 103
5.6 Worst-case CCG compression with random timing . . . 103
5.7 Worst-case CCG compression with random timing and large time deviation bounds . . . 104
5.8 Worst-case CCG compression with random call pattern and random timing . . . 104
5.9 Ratio R_{nodes} vs. time deviation abs (small scale compression) . . . 106
5.10 Ratio R_{memory} vs. time deviation abs (small scale compression) . . . 106
5.11 Compression run-time for small scale compression vs. time deviation rel. . . . 106
5.12 Ratio R_{nodes} vs. time deviation rel (small scale compression) . . . 107
5.13 Ratio R_{memory} vs. time deviation rel (small scale compression) . . . 107
5.14 Influence of final node count n on compression run-time for small scale compression. . . 107
5.15 Ratio R_{nodes} vs. time deviation abs (large scale compression) . . . 110
5.16 Ratio R_{memory} vs. time deviation abs (large scale compression) . . . 110
5.17 Compression run-time for large scale compression vs. time deviation rel. . . . 110
5.18 Ratio R_{nodes} vs. time deviation rel (large scale compression) . . . 111
5.19 Ratio R_{memory} vs. time deviation rel (large scale compression) . . . 111
5.20 Influence of final node count n on compression run-time for large sclale compression. . . 111
5.21 Absolute node count depending on maximum branching factor b. . . . 112
5.22 Compression ratios R_{nodes} and R_{memory} vs. maximum branching factor b . . . 112
5.23 Compression run-time vs. maximum branching factor b. . . . 113
5.24 Influence of trace size to node compression ratio R_{nodes} . . . 114
5.25 Progressing compression ratio R_{nodes} during CCG compression . . . 114
5.26 Influence of search length parameter L to the ratio R_{nodes} and the run-time t . . . 115
5.27 Comparison of original compression, re-compression and renewed compression . . . 116
5.28 Run-time for original compression, re-compression and renewed compression . . . 117
5.29 Compression ratio R_{nodes} vs. the number of re-start steps for adaptive compression . . . 118
5.30 Run-time for adaptive compression . . . 118
5.31 Run-Time for CCG serialization and restore . . . 119

5.32 Comparison of query run-time for initial and successive queries 121
5.33 Influence of the branching factor to the run-time for initial and successive queries 122
5.34 Temporary ressource usage of the MPI send-receive matching algorithm 123
5.35 Comparison of the total memory usage with Vampir and VampirServer 126
5.36 Comparison of the trace loading time with Vampir and VampirServer 126
5.37 Comparison of the run time of cached summary queries 126

Bibliography

[AH83] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations*. Computer Science and Applied Mathematics. Academic Press, 1983. Chapter 1, Theorem 4 (7).

[AH90] Anant Agarwal and Minor Huffman. Blocking: Exploiting Spatial Locality for Trace Compaction. *SIGMETRICS Perform. Eval. Rev.*, 18(1):48–57, 1990.

[BFJ00] P.N. Brown, R.D. Falgout, and J.E. Jones. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[BHdW$^+$95] David H. Bailey, T. Harris, Rob Van der Wigngaart, William Saphir, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-010, NASA Ames Research Center, 1995.

[BHNW01] H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. Performance Otimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Proceedings of ICCS2001, San Francisco, USA*, volume 2074 of *LNCS*, page 751. Springer, May 2001.

[BKN04] H. Brunst, D. Kranzlmüller, and W.E. Nagel. Tools for Scalable Parallel Program Analysis – Vampir and DeWiz. In *Proc. of DAPSYS 2004, 5th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Budapest, Hungary, Sept 2004.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. of The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, November 2000.

[BMN08] Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Vampir and Vampir NG Source Codes. personal communications, 2005 – 2008.

[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*, volume 2858 of *LNCS*, pages 440–449. Springer, Nov 2003.

[BNM03] Holger Brunst, Wolfgang E. Nagel, and Allen D. Malony. A Distributed Performance Analysis Architecture for Clusters. In *IEEE International Conference on Cluster Computing, Cluster 2003*, pages 73–81, Hong Kong, China, Dec 2003. IEEE Computer Society.

[BNS00] H. Brunst, W. E. Nagel, and S. Seidl. Performance Tuning on Parallel Systems: All Problems Solved? In *Proceedings of PARA2000 - Workshop on Applied Parallel Computing*, volume 1947 of *LNCS*, pages 279–287. Springer, June 2000.

[Bru08] Holger Brunst. *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs*. PhD thesis, Technische Universität Dresden, 2008. ISBN 978-3-8322-6990-6.

[CALG07] Anthony Chan, David Ashton, Rusty Lusk, and William Gropp. *Jumpshot-4 Users Guide*. Mathematics and Computer Science Division, Argonne National Laboratory, July 2007. `ftp://ftp.mcs.anl.gov/pub/mpi/slog2/js4-usersguide.pdf`.

[CEP00] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain. *PARAVER Version 2.1 Tutorial*, Nov 2000.

[CEP01a] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain. *PARAVER Version 3.0 Tracefile Description*, June 2001.

[CEP01b] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain. *PARAVER Version 3.1 Reference Manual*, Oct 2001.

[CGL98] Anthony Chan, William Gropp, and Ewing Lusk. *Users Guide for MPE: Extensions for MPI Programs*. Mathematics and Computer Science Division, Argonne National Laboratory, 1998. `ftp://ftp.mcs.anl.gov/pub/mpi/mpeman.pdf`.

[CGL00] Anthony Chan, William Gropp, and Ewing Lusk. Scalable Log Files for Parallel Program Trace Data (draft). Technical report, Argonne National Laboratory, 2000.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[Cor06] Intel Corporation. Pallas HPC Division Acquisition Questions and Answers. `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/219909.htm`, 2006.

[Cor07] Intel Corp. *Intel (R) Trace Analyzer 7.1 Reference Guide*, Nov 2007. `http://www.intel.com/`, document number 318120.

[CR71] E. G. Coffman and B. Randell. Performance Predictions for Extended Paged Memories. *Acta Informatica*, 1(1):1–13, 1971.

[dKdOSB00] J. Chassin de Kergommeaux, B. de Oliveira Stein, and P.E. Bernard. Paje, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. In *Parallel Computing, Proc. of EuroPar 2000*, pages 1253–1274, Aug 2000.

[dKdOSM03] J. Chassin de Kergommeaux, B. de Oliveira Stein, and G. Mounie. Paje Input Data Format (Draft). Technical report, Laboratoire Informatique et Distribution, Grenoble, 2003. `http://www-id.imag.fr/Logiciels/paje/publications/`.

[Eck95] B. Eckel. *Thinking in C++*. Prentice-Hall Inc., 1995.

[Fal00] Rob Falgout. SMG98 Semicoarsening Multigrid Solver. Technical report, Lawrence Livermore National Laboratory (LLNL), May 2000.

[FGM$^+$01] Thomas Fahringer, Michael Gerndt, Bernd Mohr, Felix Wolf, Graham Riley, and Jesper Larsson Träff. Knowledge Specification for Automatic Performance Analysis – APART Technical Report. In *ESPRIT IV Working Group on Automatic Performance Analysis: Resources and Tools*, January 2001.

[FGS03] Karl Führlinger, Michael Gerndt, and Andreas Schmidt. Towards Automatic Performance Analysis for Large Scale Systems. In *International Workshop on Compilers for Parallel Computers*, Jan 2003.

[For95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard - Version 1.1*, June 1995. `http://www.mpi-forum.org/docs/`.

[For97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. `http://www.mpi-forum.org/docs/`.

[GC97] Gideon Glass and Pei Cao. Adaptive Page Replacement Based on Memory Reference Behavior. *SIGMETRICS Perform. Eval. Rev.*, 25(1):115–126, 1997.

[GC00] David Grove and Craig Chambers. An Assessment of Call Graph Construction Algorithms. Technical report, IBM T.J. Watson Research Center, 2000.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *ACM Conference on Object-Oriented Programming*, pages 108–124, Atlanta, Georgia, 1997.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. GProf: A Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massachusetts, 1982.

[GKP⁺07] Markus Geimer, Björn Kuhlmann, Farzona Pulatova, Felix Wolf, and Brian Wylie. Scalable Collation and Presentation of Call-Path Profile Data with CUBE. In *Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the International Conference ParCo 2007)*, pages 645–652, Jülich/Aachen, Germany, December 2007.

[GMT04] Michael Gerndt, Bernd Mohr, and Jesper Larsson Träff. Evaluating OpenMP Performance Analysis Tools with the APART Test Suite. In *Europar 2004*, August 2004.

[Gri05] David Alan Grier. *When Computers Were Human*. Princeton University Press, Feb 2005.

[GUP03] University Linz GUP. DeWiz Source Code Documentation. `http://www.gup.uni-linz.ac.at/dewiz/documentation.php`, 2003. Linz/Austria.

[GWWM06] Markus Geimer, Felix Wolf, Brian Wylie, and Bernd Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, pages 303–312. Springer, Sept 2006.

[HL91] Virginia Herrarte and Ewing Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

[JH94] E.E. Johnson and J. Ha. PDATS: Lossless Address Trace Compression for Reducing File Size and Access Time. In *Proc. IEEE International Conference on Computers and Communications*, pages 213–219, 1994.

[JJ97] Robert J. Jenkins Jr. Hash Functions for Hash Table Lookup. *Dr. Dobb's Journal*, September 1997. `http://burtleburtle.net/bob/hash/evahash.html`.

[JMC02] Guohua Jin and John Mellor-Crummey. Experiences Tuning SMG98 - a Semicoarsening Multigrid Benchmark based on the Hypre Library. In *Proceedings of the International Conference on Supercomputing*, New York, June 2002.

[JSC08] JSC (Jülich Supercomputing Centre), Jülich, Germany. *Scalasca 1.0 Documentation and Source Code*, Jun 2008. `http://www.fz-juelich.de/jsc/scalasca/software/download/`.

[KBB⁺06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science ICCS 2006: 6th International Conference*, LNCS 3992, Reading, UK, May 2006. Springer.

[KBB08] Andreas Knüpfer, Holger Brunst, and Ronny Brendel. *Open Trace Format Specification and Source Code*, June 2008. `http://www.tu-dresden.de/zih/otf/`.

[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 Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *"Tools for High Performance Computing", Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, Stuttgart, Germany, July 2008. Springer-Verlag.

[KBN05] Andreas Knüpfer, Holger Brunst, and Wolfgang E. Nagel. High Performance Trace Visualization. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 258–263, Lugano, Switzerland, februar 2005. ISBN 0-7695-2280-7.

[KDS99] K. Kevin Dowd and C. Severance. *High Performance Computing (RISC Architectures, Optimization & Benchmarks), 2nd ed.* O'Reilly Media, 1999.

[KGV96a] D. Kranzlmüller, S. Grabner, and J. Volkert. Event Graph Visualization for Debugging Large Applications. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT 96)*, pages 108–117, Philadelphia, Pennsylvania, USA, May 1996.

[KGV96b] Dieter Kranzlmüller, Siegfried Grabner, and Jens Volkert. Debugging Massively Parallel Programs with ATEMPT. In *Proc. of High-Performance Computing and Networking (HPCN Europe)*, pages 806–811, Brussels, Belgium, 1996.

[KKN04] Andreas Knüpfer, Dieter Kranzlmüller, and Wolfgang E. Nagel. Detection of Collective MPI Operation Patterns. In J. Dongarra D. Kranzlmüller, P. Kacsuk, editor, *Recent Advances in PVM and MPI, Proc. of the 11th European PVM/MPI Users Group Meeting (EuroPVM/MPI 2004)*, volume 3241 of *LNCS*, pages 259–267. Springer, 2004.

[KL94] Edward Karrels and Ewing Lusk. Performance Analysis of MPI Programs. In Jack Dongarra and Bernard Tourancheau, editors, *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, pages 195–200. SIAM Publications, 1994.

[Knu97] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, July 1997.

[Knu98] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

[Knü03] Andreas Knüpfer. A New Data Compression Technique for Event Based Program Traces. In *Proccedings of ICCS 2003 in Melbourne/Australia*, volume LNCS 2659/2003, pages 956 – 965. Springer, June 2003.

[koj05] The Kojak Source Distribution, Version 2.1.1. `http://www.fz-juelich.de/zam/kojak/`, 2005.

[Kra00] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Joh. Kepler University Linz, September 2000. `http://www.gup.uni-linz.ac.at/~dk/thesis`.

[Kra02] Dieter Kranzlmüller. Communication Pattern Analysis in Parallel and Distributed Programs. In *Proc. of the 20th International Multi-Conference of Applied Informatics (AI 2002), International Symposia on Software Engineering, Data-bases, and Applications, International Association of Science and Technology for Development (IASTED)*, pages 153–158, Innsbruck, Austria, February 2002. ACTA Press.

[KSV03] Dieter Kranzlmüller, Michael Scarpa, and Jens Volkert. DeWiz - A Modular Tool Architecture for Parallel Program Analysis. In *Euro-Par 2003 Parallel Processing, Proc. 9th International Euro-Par Conference*, Springer LNCS 2790, pages 74–80, Klagenfurt, Austria, August 2003.

[KSW99] Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson. Trace Reduction for Virtual Memory Simulations. In *Measurement and Modeling of Computer Systems*, 1999.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, pages 558 – 565, July 1978.

[LAN] Advanced Simulation and Computing Program. `http://www.lanl.gov/asci/`.

[lGA02] Jean loup Gailly and Mark Adler. *Zlib 1.1.4 Manual*, March 2002. `http://www.zlib.net/manual.html`.

[LGP$^+$96] Jesus Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A Parallel Program Development Environment. In *Proc. of 2nd International EuroPar Conference (EuroPar 96)*, Lyon/France, August 1996.

[MCC$^+$95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer, Special Issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems*, 28(11), Nov 1995.

[MDG$^+$04] J. Michalakes, J. Dudhia, D. Gill, Thomas B. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Reseach and Forecast Model: Software Architecture and Performance. In George Mozdzynski, editor, *11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, Reading, U.K., October 2004.

[MKG$^+$04] Matthias S. Müller, Kumaran Kalyanasundaram, Greg Gaertner, Wesley B. Jones, Rudolf Eigenmann, Ron Liebermann, Matthijs van Waveren, and Brian Whitney. SPEC HPG Benchmarks for High Performance Systems. *International Journal of High Performance Computing and Networking.*, 1(4):162–170, 2004.

[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 Christian Bischof, Martin Bücker, Paul Gibbon, Gerhard Joubert, Thomas Lippert, Bernd Mohr, and Frans Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007. ISBN 978-1-58603-796-3.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, April 1965.

[MSM05] A. D. Malony, Sameer S. Shende, and Alan Morris. Phase-Based Parallel Performance Profiling. In *Proc. of PARCO 2005 Conference*, Malaga, Spain, September 2005.

[MWD00] Robert Muth, Scott Watterson, and Saumya Debray. Code Specialization based on Value Profiles. In *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, volume 1824 of *Springer LNCS*, pages 340–359, June 2000.

[MWD$^+$05] Shirley Moore, Felix Wolf, Jack Dongarra, Sameer Shende, Allen Malony, and Bernd Mohr. A Scalable Approach to MPI Application Performance Analysis. In B. Di Martino, editor, *Proc. of EuroPVM/MPI 2005*, Springer LNCS 3666, pages 309–316, Sorrento, Naples, Italy, Sept 2005.

[NA] W.E. Nagel and A. Arnold. Performance Visualization of Parallel Programs - the PARvis Environment. In *Proceedings of Intel Supercomputer Users Group (ISUG) Conference*.

[NAW$^+$95] W.E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. In Jack J. Dongarra, Hans-Werner Meuer, and Erich Strohmaier, editors, *TOP500 Supercomputer Sites*, Nov 1995.

[Nic96] O. Nickolayev. Performance Data Reduction Using Dynamic Statistical Clustering. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1996.

[NMSdS07] M. Noeth, F. Müller, M. Schulz, and B.R. de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *Proc. of International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, Mar 2007.

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

[Rab00] Rolf Rabenseifner. *The Controlled Logical Clock, a Global Clock for Trace-Based Monitoring of Parallel Applications*. PhD thesis, Universität Stuttgart, Fakultät Informatik, 2000. `http://opus.uni-stuttgart.de/opus/volltexte/2000/600/`.

[RAM03] P.C. Roth, D.C. Arnold, and B.P. Miller. MRnet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proc. of Supercomputing 2003*, Washington, DC, USA, Nov 2003. IEEE Computer Society.

[RMdSS08] P. Ratn, F. Müller, B.R. de Supinski, and M. Schulz. Preserving Time in Large-Scale Communication Traces. In *Proc. of the 2008 ACM International Conference on Supercomputing*, Island of Kos, Greece, June 2008.

[Rot95] P.C. Roth. ETRUSCA: Event Trace Reduction using Statistical Data Clustering Analysis. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Dec 1995.

[Sam89] A. D. Samples. Mache: No-Loss Trace Compaction. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 89 – 97, Oakland, California, United States, 1989. ISSN:0163-5999.

[SGH06] Martin Schulz, Jim Galarowicz, and William Hachfeld. Open|SpeedShop: Open Source Performance Analysis for Linux Clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 14, New York, NY, USA, 2006.

[She] S. Shende. TAU File Type Conversion Diagram. `http://www.cs.uoregon.edu/research/tau/docs.php`.

[She04] Sameer Shende. TAU Source Code, Version 2.13.5. personal communications, 2004.

[SM05] S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications (ACTS Collection)*, 2005.

[Smi77] A. J. Smith. Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Trans. Softw. Eng.*, 3(1):94–101, 1977.

[STF07] Intel GmbH, Brühl, Germany. *Intel Trace Collector User's Guide*, 2007. document number 318119, `http://www.intel.com/`.

[UOR] Performance Research Lab, University of Oregon. `http://www.nic.uoregon.edu/prl/home.php`.

[Vel95a] Todd Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[Vel95b] Todd L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[Voi06] Bernhard Voigt. Effiziente Erkennungs- und Visualisierungsmethoden für hierarchische Trace-Informationen. Diploma Thesis (German), TU Dresden, 2006.

[WBS$^+$00] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proc. of SC2000: High Performance Networking and Computing*, Dallas, TX, USA, Nov 2000.

[Wel84] Terry Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17:8–19, June 1984.

[WM98] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. Technical report, Forschungszentrum Jülich GmbH, April 1998. FZJ-ZAM-IB-9803.

[WM99] Felix Wolf and Bernd Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN)*, Springer LNCS 1593, pages 503–512, Amsterdam, The Netherlands, April 1999.

[WM00a] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of MPI Applications Based on Event Traces. In *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, Springer LNCS 1900, pages 123–132, Munich, Germany, 2000.

[WM00b] Felix Wolf and Bernd Mohr. EARL - Language Reference. Technical report, ZAM, FZ Jülich, Germany, 2000. FZJ-ZAM-IB-2000-01.

[WM00c] Felix Wolf and Bernd Mohr. Specifying Performance Properties Using Compound Runtime Events. Technical Report FZJ-ZAM-IB-2000-10, ZAM, FZ Jülich, August 2000.

[WM01] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of SMP Cluster Applications. Technical Report FZJ-ZAM-IB-2001-05, ZAM, FZ Jülich, August 2001.

[WM04] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[WMDM07] Felix Wolf, Bernd Mohr, Jack Dongarra, and Shirley Moore. Automatic Analysis of Inefficiency Patterns in Parallel Applications. *Concurrency and Computation: Practice and Experience*, 2007.

[Wol04] F. Wolf. EARL - API Documentation. Technical report, University of Tennessee, October 2004. Technical Report ICL-UT-04-03.

[ZAM] Central Institute for Applied Mathematics (ZAM), Research Centre Jülich. `http://www.fz-juelich.de/zam/`.

[ZL77] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, pages 75–81, may 1977.

[ZLGS99] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.