Concepts for In-memory Event Tracing
Runtime Event Reduction with Hierarchical Memory Buffers

Michael Wagner

Abstract — High Performance Computing (HPC) systems are getting more and more powerful but more and more complex, as well. Supportive environments, such as performance analysis tools, are essential to assist developers in utilizing the tremendous computational resources of such complex systems. For long-running and large-scale parallel applications, event-based performance analysis faces three challenges, yet unsolved: the number of resulting trace files limits scalability, the huge amount of collected data overwhelms file system and analysis capabilities, and the measurement bias, in particular, due to intermediate memory buffer flushes prevents a correct and meaningful analysis.
This thesis proposes concepts for an in-memory event tracing workflow to meet these challenges. These concepts include new enhanced encoding techniques to increase memory efficiency, novel strategies for runtime event reduction to dynamically adapt trace size during runtime, and the Hierarchical Memory Buffer data structure, which incorporates a multi-dimensional, hierarchical ordering of events by common metrics, such as time stamp, calling context, event class, and function call duration. These concepts allow a trace size reduction of up to three orders of magnitude and can keep an entire measurement within a single fixed-size memory buffer, while still providing a coarse but meaningful analysis of the application.

1 Introduction

High Performance Computing (HPC) systems provide enormous computational resources to support large-scale simulations in leading-edge scientific research, such as climate and weather prediction, chemistry, applied physics, or DNA and cancer research. Today, High Performance Computing typically includes a large number of processing elements working jointly on a computationally intensive problem. For the next milestone, exa-scale supercomputers capable of $\mathcal{O}(10^{18})$ floating point operations per second, this approach is very likely to persist. Writing software for systems of this scale is demanding and involves hybrid and new programming models, accelerated computing, and energy considerations. Hence, appropriate supporting tools, such as debuggers and performance analyzers, are inevitable to develop applications that utilize the tremendous capabilities of current and future HPC systems.
Performance analysis tools assist developers not only in identifying performance issues in their applications but also in understanding their behavior on complex and heterogeneous systems. The subset of event tracing tools gathers information about the behavior of an application during runtime by recording runtime events, for instance, entering and leaving a function. Event tracing records each event of a parallel application in detail and allows an exact reconstruction of the application behavior. Thus, it enables capturing the dynamic interaction between thousands of concurrent processing elements and the identification of outliers from the regular behavior. Such detail comes with the cost that event-based tracing frequently results in huge data volumes, even though single events are rather small.
In fact, the large amount of collected data, in particular, for massively parallel or long running applications results in three challenges, yet unsolved: First, the number of resulting trace files limits scalability since the collected data is usually stored in one file per processing element. While HPC parallel file systems are highly optimized for data throughput, the simultaneous creation of hundreds of thousands or even millions of event tracing files overwhelms any parallel file system. Second, the aggregated size of the resulting trace files quickly swallows up storage capacities and overstrains analysis capabilities. Third, the bias caused by intermediate memory buffer flushes. Recorded event data is typically buffered before it is written to the file system to reduce expensive file system interactions. Whenever such an internal memory buffer is exhausted, the content is transferred to the file system; usually in an unsynchronized fashion. Such uncoordinated intermediate memory buffer flushes during a measurement introduce extensive bias and lead to a falsification of the recorded program behavior, which prevents a correct and meaningful analysis.

1.1 In-memory Event Tracing

An *in-memory event tracing workflow* that completely omits file system interaction would meet these challenges. First, keeping recorded event data in main memory for the complete measurement would not only bypass the limitations in the number of file handles but it would eliminate the overhead of file creation, writing and reading altogether. Second, tracing data that could be kept in main memory results in remarkably smaller data volumes. Third, an in-memory workflow would exclude the bias caused by non-synchronous intermediate memory buffer flushes during a measurement run.
But there is one catch. Keeping event data in main memory for a complete measurement requires that recorded data fits into a single memory buffer of an event tracing library. Unfortunately, measurement runs may collect hundreds of megabytes up to tens or hundreds of gigabytes of data per processing element. To make things worse, the part of the main memory left to store the data is rather small, since most applications utilize main memory intensively. This thesis applies to the challenge of fitting an entire measurement of arbitrary length and scale into a single fixed-sized memory buffer for each processing element and, therefore, setting the premise for an in-memory event tracing workflow.

1.2 Contribution of this Thesis

The contributions of this thesis are novel concepts to enable an in-memory event tracing workflow. These concepts are divided in two central parts:
Enhanced encoding techniques and *strategies for event reduction* that dynamically adapt trace size during runtime to the given memory allocation form the first central part of the contributions of this thesis. The combination of both allows to keep the data of an entire measurement within a single fixed-sized memory buffer and, therefore, enable an in-memory event tracing workflow.
The *Hierarchical Memory Buffer* is the second central contribution of this thesis. The Hierarchical Memory Buffer is a new data structure that uses hierarchy information, such as time stamp, calling context, event class, and function call duration, to presort events. It allows to perform the aforementioned event reduction operations with minimal overhead. Furthermore, such a hierarchy-based event representation allows new event filter operations, unfeasible with a traditional flat, continuos memory buffer layout. Such a new filter method is a filtering based on the duration of function calls, which eliminates all short-running functions while keeping outliers important for performance analysis. In addition, several typical analysis requests can benefit from a hierarchy-aided traversal of recorded event data.

2 Concepts for In-memory Event Tracing

The main challenge for an in-memory event tracing workflow is to keep events of an entire measurement within a single fixed-sized memory buffer. Most event tracing approaches buffer recorded events in main memory to reduce file system interaction. Once the memory buffer is exhausted, the measurement is either aborted or event data is flushed to a file. An in-memory event tracing workflow, however, must avoid both of the above. Therefore, an in-memory workflow must apply methods to reduce the amount of data in the memory buffer by either reducing the number of events or the amount of memory per event. Main constraints are that, first, these methods introduce minimal overhead to avoid additional measurement perturbation, second, the measurement can contain an arbitrary but finite amount of events, and, third, the memory buffer can be of arbitrary but fixed size.
The methods presented in this thesis can be categorized into three major steps within an in-memory event tracing workflow: *selection and filtering*, *encoding and compression*, and *event reduction*.

2.1 Selection and Filtering

The first step contains methods to either select events for monitoring before the measurement starts or filter events during runtime. It includes a novel approach that filters functions based on their actual runtime duration. This way, highly frequent calls to short-running functions, e.g., set and get class methods or tiny helper functions, are effectively filtered while still keeping outliers that impact the application behavior. This is covered in detail in Section 3.2 and 3.4.4 of the thesis.

2.2 Enhanced Encoding Techniques

The second step is an efficient storage of event tracing data within the memory buffer. This requires a compact encoding and low-overhead compression. This section presents new enhanced techniques to remarkably increase memory efficiency. This step builds on existing methods in the Open Trace Format 2 [5], which is similar to many other event trace formats, such as the Paraver Trace Format [4], the TAU trace format [14], as well as its two predecessors the Open Trace Format [8] and Epilog [15].
At first, this section describes the basic memory representation of an event record in a binary encoding. In a basic memory representation of an event record in a binary encoding, an event record consists of three main parts: First, a record token that defines the type of an event, e.g., entering or leaving a code region, sending or receiving a message; second, an exact time stamp telling when the event occurred; and third, event specific attributes, e.g., a region ID for a region enter record. Figure 1 shows a generic event record with two attributes in the basic record design of the Open Trace Format 2.

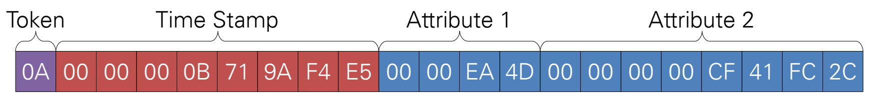


Figure 1: Basic memory representation of event records. First, a one-byte record token defining the type of event, followed by a time stamp of eight bytes telling when the event occurred; and third, event specific attributes, one with four bytes and one with eight bytes.

Based on this basic memory representation the thesis introduces five enhanced encoding techniques to reduce memory allocation:

1. *Splitting of Timing Information and Event Data*: storing timing information and event data separately eliminates redundant timing information for consecutive events with identical time stamps.
2. *Leading Zero Elimination*: omitting leading zero bytes in integer values for all event attributes since the majority of values is much smaller than the hypothetical maximum.
3. *Delta Encoding*: some monotonic increasing event attributes begin with a very high offset, such as time stamps and hardware performance counters. In this case, storing only the difference (delta) to the previous value leads to much smaller values to store.
4. *Token Encoding*: for frequent events, such as enter/leave events and performance counters, small values are directly encoded in the token.
5. *Timer Resolution Reduction*: monitoring tools usually use timer source with a very high resolution (CPU cycles) while event frequency is the order of microseconds. Thus, it is possible to reduce the stored timer resolution without perceivably degrading the accuracy of the timing information.

2.3 Event Reduction

While these first two steps can significantly reduce memory allocation, they fail the most important criterion for an in-memory event tracing workflow: they cannot guarantee that the data of an arbitrary measurement fits into a single memory buffer of fixed size. Consequently, the third and last step is entirely different. It is triggered whenever the memory buffer is exhausted; typically this is the point where the memory buffer is either flushed to a file or the measurement is aborted. The crucial point is making memory space available again by reducing the events already stored within the memory buffer while in the same time introducing minimal overhead. This section introduces four strategies for event reduction: a reduction by the order of occurrence of the events, by their event class, by the current calling depth, and by the duration of a code region.
Each strategy is reviewed based on two criteria. First, the quality of the remaining information. Since performance analysis has two major goals, to better understand an application's behavior and to identify potential performance issues, the comparison is based on how good these goals can still be achieved with the reduced event set. Second, the granularity of the individual event reduction operation. In this respect, granularity means the amount of data that is discarded in a single event reduction operation. If the reduction steps are too large, a lot of information might unnecessarily be discarded.

2.3.1 Reduction by Order of Occurrence

The first strategy is to reduce events by their order of occurrence. This means that events are either discarded or kept depending on the time they occurred. If the memory buffer is capable to store n events, there are three different ways this method can be applied: First, store the first n events, i.e., recording is stopped once the memory buffer is exhausted. Second, store the last n events. This method requires a cyclic buffer that starts overwriting events in the front of the buffer whenever the end of the buffer is reached. Third, store either the first or last n events within a specific application phase.

These methods provide the complete application behavior within the recorded interval; either at the beginning, at the end, or in the middle of an application, depending on which method is chosen. Outside of this application interval, there is no information about the application's behavior available because all according events are discarded. Thus, a performance analysis based on these methods allows a good understanding about the recorded interval of the application but cannot provide any information about the part that was discarded. The same applies for the ability to detect performance issues. Therefore, the quality of the overall information about an application strongly depends on the structure of the application and whether the right interval is selected for recording. The granularity of a reduction by the order of occurrence is very high; reduction steps are in the order of a few events.

2.3.2 Reduction by Event Class

Single events that are recorded can be categorized into different classes of events, e.g., entering and leaving a code region, point-to-point or collective communication, performance metrics like hardware performance counters, or I/O operations. Not all of these different event classes are of same importance when analyzing an application. For instance, for an analysis of the communication behavior, communication events are very important while specific hardware performance counters, like cache misses, are less important. For an analysis of single thread performance vice versa. Hence, it is possible to order the different event classes and start event reduction with the least important event class.

In contrast to the first strategy that provides complete information within the recorded interval, this approach provides information for the entire application interval. However, the information is restricted to the events of those event classes that remain in the trace. Hence, the remaining events allow a partial performance analysis. Analyzing the remaining events results in a good understanding of those aspects of the application that are represented by the remaining event classes. About application behavior deducible by events of discarded event classes, no knowledge can be obtained. The detection of performance problems shares the same restrictions. In this respect, the quality of the overall information about an application's behavior depends on an appropriate order of event classes by their importance.

The granularity of this reduction operation relies on the distribution of the different event classes by their memory allocation. Unfortunately, the statistical survey of the reviewed applications and kernels shown in Figure 2 reveals that there are only three dominating event classes: enter/leave events, performance metrics, and point-to-point communication.

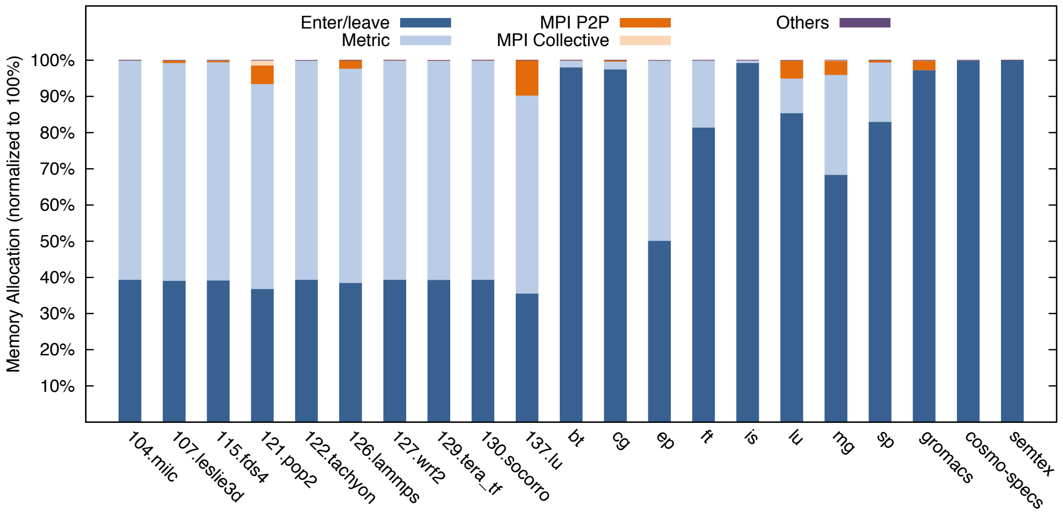


Figure 2: Distribution of event classes by size of memory allocation.

2.3.3 Reduction by Calling Depth

Next to an order by event class, events can also be ordered by their calling depth. The third event reduction strategy uses this order and starts reduction with those events on the deepest call stack level which implies that no further events on this or a deeper call stack level are recorded. This strategy is based on the assumption that events on the deepest call stack level usually contribute less to the overall understanding of the application behavior than those on higher levels. Still, these events may be the source for a performance issue.
Similar to the second strategy, the behavior and potential performance issues can only be fully reconstructed with the events in the remaining call stack levels. However, while the first two strategies completely discard the information with the events that carry them, this strategy allows to obtain parts of the information from higher call stack levels. In particular, when reducing the call stack level that contains the events that mark a performance issue, the actual cause of the performance issue is lost. Yet, a performance analysis might still allow to recognize the impact of this performance issue in the remaining call stack levels. Figure 3 demonstrates this correlation between cause and impact for an exemplary load imbalance in a basic timeline visualization.

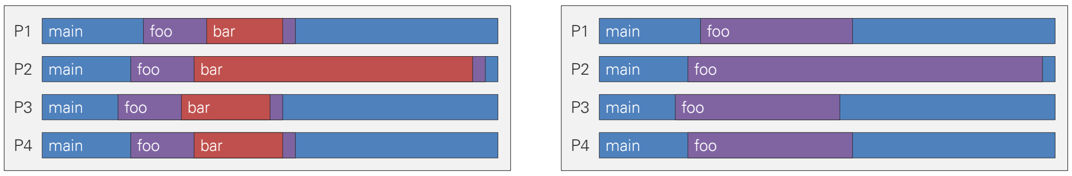


Figure 3: Correlation between cause and impact of a performance issue in a timeline visualization.

The complete event trace shows a load imbalance caused by the function *bar* on process two. When the event reduction by call stack level engages, the deepest call stack level containing *bar* is reduced and the cause of the performance issue cannot be identified anymore. However, the impact of the performance issue and, therefore, the performance issue itself is still detectable. Hence, the knowledge gained about an application's behavior and the ability to detect performance issue is reduced but not completely lost for individual aspects. Of course, the impact of the performance issue becomes more and more blurred when further call stack levels are discarded. In the exemplary load imbalance, the performance issue is completely lost if the second deepest call stack level, containing *foo*, is eliminated, as well.
The second criteria, the granularity of single reduction steps, of this strategy strongly depends on an application's structure with regard to its call stack level distribution. An ideal case is a deep and equally distributed call stack, which allows a reduction in very fine grained steps. Figure 4 shows the call stack distribution for two selected applications.

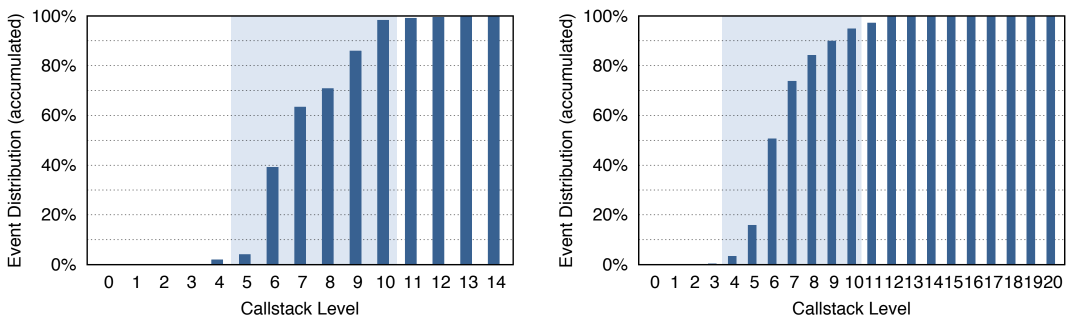
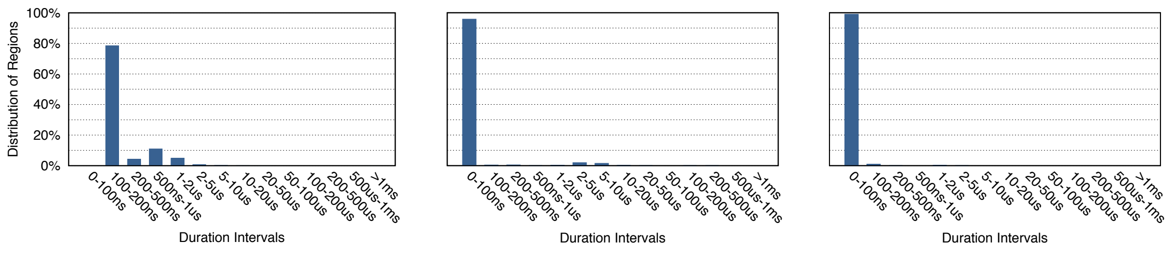


Figure 4: Callstack distribution for Gromacs (left) and Semtex (right).

2.3.4 Reduction by Duration

The fourth and last strategy uses the duration of code regions as criterion for event reduction. Having in mind that enter/leave events are the most dominant event class, next to performance metrics if they are recorded, the class of enter and leave events provides a good starting point for a further, more detailed reduction strategy. For instance, recording every function call with the same detail is prone to fail,

especially, when tiny and often-used functions are monitored, e.g., small helper functions or get and set class methods. An event reduction by duration addresses this impact of high-frequency function calls and presents a method to minimize the amount of high-frequency function calls while still keeping outliers that have an impact on an application's behavior.
Automated instrumentation techniques like compiler instrumentation are most convenient and easy-to-use. Hence, many event-based monitoring tools use such automated techniques as the default to define events [9, 11, 6, 14, 3]. One side effect is, that compiler instrumentation prevents the inlining[1] of tiny and short-running functions, such as small helper functions or get and set class methods. By itself, a suppressed inlining and recording of these functions provides tools an opportunity to record and analyze an application's behavior very detailed. However, if such short-running functions are heavily called they might overwhelm the capacity of the recording memory buffer while at the same time contribute very little to the overall application behavior.
Figure 5 presents the results of an statistical survey with some selected applications and kernels that evaluates the distribution of function calls depending on their duration. While the definite distribution deviates slightly, depending on the number of locations and the problem size, the majority of applications shows a clear trend. All applications, except *ft*[2] from the NAS Parallel Benchmarks, use short-running function calls at a very high frequency.



(a) Gromacs (b) Cosmo-specs-fd4 (c) NAS Parallel Benchmarks, is

Figure 5: Duration distribution for selected applications and kernels.

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

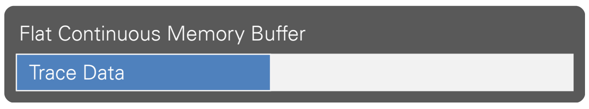
3 The Hierarchical Memory Buffer

This section introduces the Hierarchical Memory Buffer, a novel data structure that allows to perform the aforementioned event reduction operations with minimal overhead. The presented event reduction strategies require an efficient identification and elimination of events that are already stored in the memory buffer. However, currently non of the existing event tracing tools and libraries supports such an efficient elimination of events. They all use a flat continuous memory buffer that, although, allowing the elimination of events already stored in the memory buffer, introduces an enormous overhead when engaged.

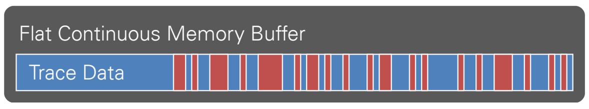
[1]Inline expansion or inlining is a compiler optimization that replaces a function call site with the body of the callee, which usually results in improved time and space usage at runtime.

[2]With regard to the complete set of applications in the thesis, Section 3.4.4

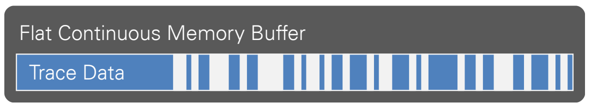
A flat continuous memory buffer stores the recorded events in the order they occurred until the memory buffer is exhausted (see Figure 6(a)). When the memory buffer is exhausted the event reduction is triggered. Since all events are scattered over the memory buffer, the entire memory buffer needs to be scanned to find all events that match the criterion for reduction (see Figure 6(b)), e.g., all events of the deepest call stack level for a reduction based on the calling depth. When all events matching the reduction criteria are found, they are discarded and the according memory sections are marked as free (see Figure 6(c)). Since events occur at a high frequency and are typically only a few bytes small, there are plenty of small free sections scattered over the whole memory buffer. This leaves a highly fragmented memory buffer that cannot be used for writing further events. Thus, all non-free memory sections need to be moved to collapse the fragmented memory buffer to a single continuous memory segment that leaves a continuous free memory section at the end to store further events (see Figure 6(d)).



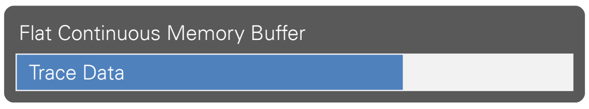
(a) Collecting events until the memory buffer is filled.



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



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

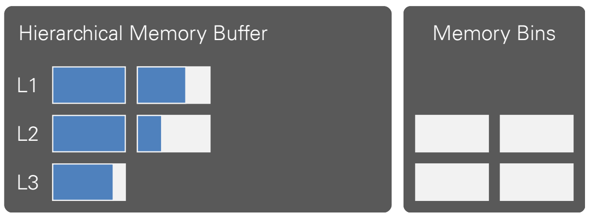


(d) The remaining non-free memory is collapsed to one memory section to provide a continuous free memory section at the end for new events.

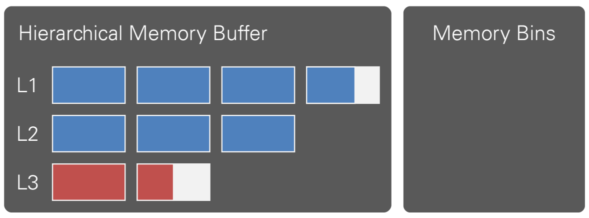
Figure 6: Event reduction with a flat continuous memory buffer.

The computational complexity of the reduction operation is in $\mathcal{O}(n)$, with n being the number of stored events. Since a memory buffer, depending on its size, can contain several million events such a reduction operation introduces a remarkable overhead when using a traditional flat continuous memory representation.

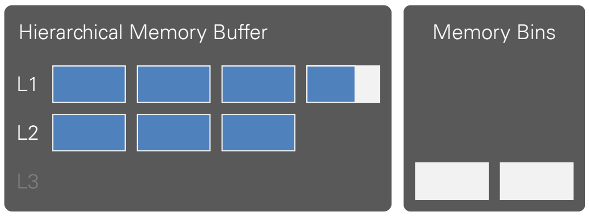
In contrast to flat continuous memory buffer, the Hierarchical Memory Buffer is organized as a multi-dimensional array, where each *hierarchy dimension* represents one possible hierarchical order with a flexible number of different values within that hierarchical order, called *hierarchy levels*. In the context of event reduction, for instance, one dimension can represent the calling depth and another the event class. Instead of one huge memory chunk, the total memory allocation for the according memory buffer is divided in plenty of small memory sections, called *memory bins*. These memory bins can be dynamically distributed to any hierarchy level in each dimension. Whenever an event needs to be stored at a certain hierarchy level and there is either no memory bin assigned or the current memory bin is exhausted, a free memory bin is distributed to this hierarchy level.
Figure 7 demonstrates the event reduction with such a hierarchical event representation. Again, for simplification this example considers at first only one reduction criterion, e.g., the calling depth. Thus, the according memory buffer's layout is an one-dimensional array. When the first event needs to be stored, usually on call stack level $L1$, no memory bin has been assigned to this hierarchy level, so far. Thus, the memory buffer checks if there is a free memory bin available, which is true in this case, and one memory bin is assigned to the hierarchy level $L1$, so, the event can be stored. If an event needs to be stored on a different hierarchy level, a free memory bin is assigned the same way. The same applies, when on any hierarchy level the current memory bin is exhausted. After some time, this leads to a situation like in Figure 7(a): Five memory bins are assigned to the hierarchy levels $L1$ - $L3$ and four free memory bins are available. Hence, four additional memory bins can be assigned to the hierarchy levels. After that, all memory bins are assigned and there are no free memory bins available anymore. This leads to the situation in Figure 7(b): An event needs to be stored at the hierarchy level $L2$ but there are no free memory bins available. At this point, the event reduction is triggered and all events of a certain hierarchy level are discarded, for instance, all events of the deepest call stack level, in this case, level $L3$. Since all events are already sorted by their call stack level the event reduction operation can be done with



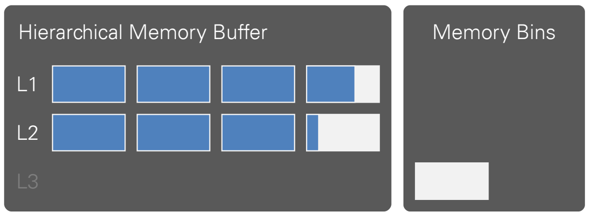
(a) Collecting events until the memory buffer is filled. Whenever a memory bin is filled a free one is assigned



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



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



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

Figure 7: Event reduction with the Hierarchical Memory Buffer.

minimal costs. The event reduction just revokes all memory bins assigned to the hierarchy level $L3$ and adds them again to the pool of free memory bins. In addition, the hierarchy level $L3$ is marked as closed, so, all future events on this hierarchy level are discarded right away. After that, the two revoked memory bins are available again (see Figure 7(c)). Therefore, one of them can be assigned to the hierarchy level $L2$ and the event that triggered the event reduction can be stored (see Figure 7(d)).
This way, the computational complexity of the reduction operation is reduced to be in $\mathcal{O}(b)$, with b being the number of memory bins to revoke. Next to the layout as one-dimensional array as for the example above, the hierarchical event representation can be organized as a multi-dimensional array, as well. In that case, the event reduction can be applied on a complete row or column within the multi-dimensional array. This way, a hierarchical memory representation is able to support all event reduction techniques simultaneously.
The Hierarchical Memory Buffer data structure and its construction, as well as the application of event reduction with the Hierarchical Memory Buffer and the adaption of common analysis techniques is discussed in detail in the thesis, Sections 4.2 - 4.5.

4 Evaluation

The evaluation chapter in the thesis presents an evaluation of the enhanced encoding techniques and the Hierarchical Memory Buffer data structure including its capabilities to support the event reduction strategies. In addition, a detailed case study demonstrates the benefits of the combined approach for a real-life application. However, this section only highlights the increased memory efficiency of the enhanced encoding techniques and a short summary of the case study.

4.1 Enhanced Encoding Techniques

The following measurement compares the prototype, called OTFX, which includes all presented enhanced encoding techniques, with other well-established trace formats and libraries. These encoding enhancements proof to be effective and reduce memory allocation during runtime by a factor of 3.3 to 7.2 (see Figure 8) for the evaluated applications and application kernels while in the same do not introduce any additional overhead on the tracing library. In fact, the overhead is slightly reduced since less memory must be allocated. With these results the enhanced encoding techniques outperform the memory efficiency of well-established general purpose compression libraries, which introduce a five times higher overhead. Therefore, the enhanced encoding techniques provide a remarkable improvement to existing event trace formats.

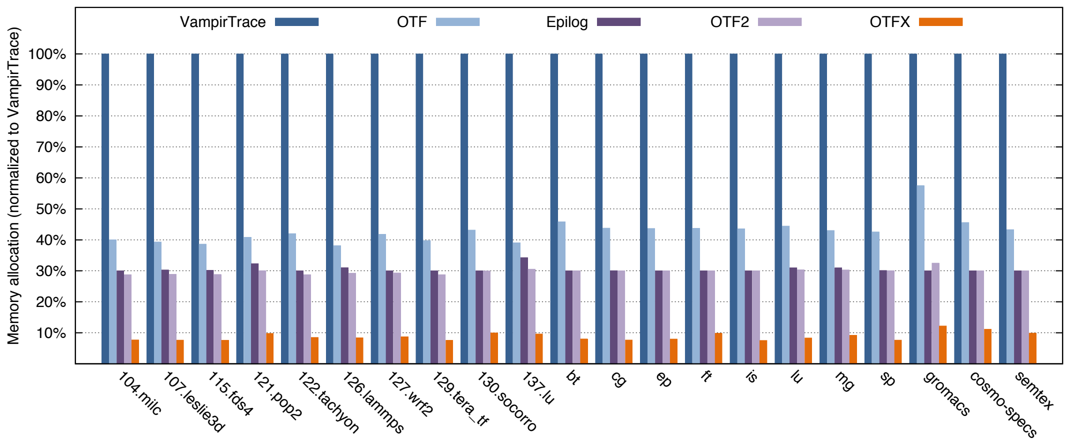
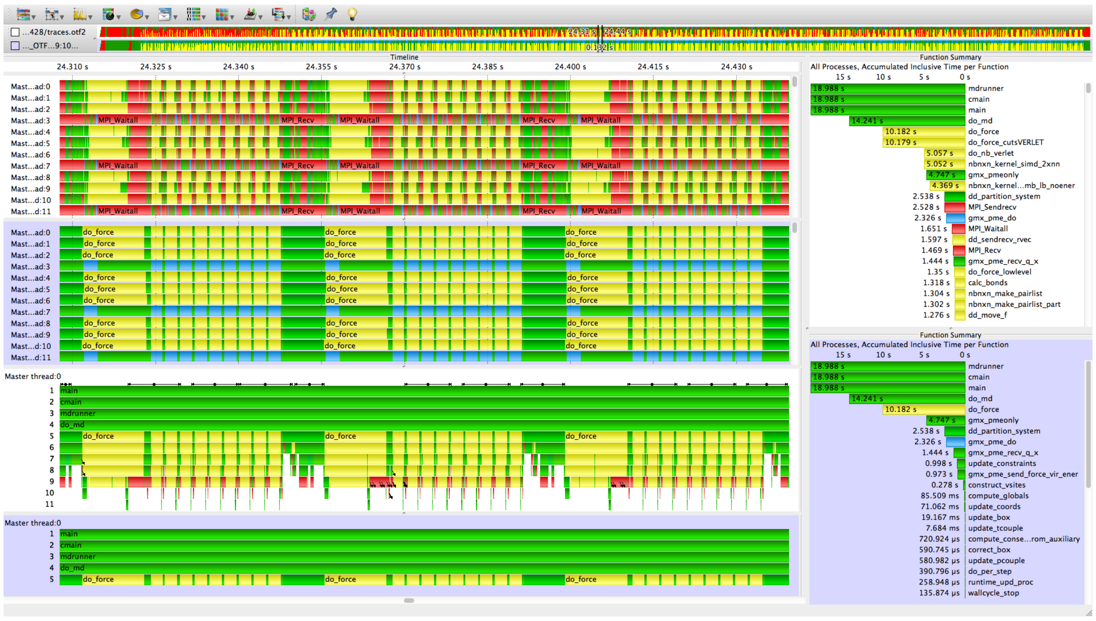


Figure 8: Memory allocation for different event trace data formats.

4.2 Case Study

The case study applies the concepts of the thesis to the molecular dynamics package Gromacs [7]. The combined concepts keep an measurement estimated with about 70 GiB of data per process within a single memory buffer of 100 MiB. This allows a measurement and meaningful analysis of a complete production run, infeasible before. The study demonstrates that the enhanced encoding techniques combined with the event reduction presented in this thesis can remarkably reduce the resulting trace size up to three orders of magnitude, while still providing a coarse but meaningful analysis of the application.
Figure 9 presents a screenshot of a visual performance analysis of Gromacs with Vampir [13]. The figure visualizes the prominent functions on calling depth five *do_force* and *gmx_pme_do* in yellow and blue, respectively, the rest of the application functions in green and MPI communication in red. First, it can be seen that the remaining events of the reduced version equal exactly the complete version, which is highlighted by the timeline view, as well as the function summary. Second, any analysis option related to MPI communication is impossible, due to the reduction of all event classes except enter/leave. Third, all functions with a calling depth bigger than five are not contained in the reduced trace resulting in a considerably lower detail, which can be seen in the timeline view, as well as the process timeline of process zero. However, the reduced trace clearly identifies the overall program behavior. It illustrates the function decomposition within each group of four processes and the iterative blocks of the application.



Function Summary

All Processes, Accumulated Inclusive Time per Function

18.988 s	mdrunner
18.988 s	cmain
18.988 s	main
14.241 s	do_md
10.182 s	do_force
10.179 s	do_force_cutsVERLET
5.057 s	do_nb_verlet
5.052 s	nbnxn_kernel_simd_2xnn
4.747 s	gmx_pmeonly
4.369 s	nbnxn_kernel...mb_lb_noener
2.538 s	dd_partition_system
2.528 s	MPI_Sendrecv
2.326 s	gmx_pme_do
1.651 s	MPI_Waitall
1.597 s	dd_sendrecv_rvec
1.469 s	MPI_Recv
1.444 s	gmx_pme_recv_q_x
1.35 s	do_force_lowlevel
1.318 s	calc_bonds
1.304 s	nbnxn_make_pairlist
1.302 s	nbnxn_make_pairlist_part
1.276 s	dd_move_f

Function Summary

All Processes, Accumulated Inclusive Time per Function

18.988 s	mdrunner
18.988 s	cmain
18.988 s	main
14.241 s	do_md
10.182 s	do_force
4.747 s	gmx_pmeonly
2.538 s	dd_partition_system
2.326 s	gmx_pme_do
1.444 s	gmx_pme_recv_q_x
0.998 s	update_constraints
0.973 s	gmx_pme_send_force_vir_ener
0.278 s	construct_vsites
85.509 ms	compute_globals
71.062 ms	update_coords
19.167 ms	update_box
7.684 ms	update_tcouple
720.924 μs	compute_conse...rom_auxiliary
590.745 μs	correct_box
580.982 μs	update_pcouple
390.796 μs	do_per_step
258.948 μs	runtime_upd_proc
135.874 μs	wallcycle_stop

Figure 9: Event trace visualization with Vampir without (top, white background) and with the concepts of this thesis (bottom, blue background) zoomed to approximately three iterations.

5 Conclusion

This thesis emphasizes the benefits of an in-memory workflow for event-based performance monitoring and presents concepts to realize such a workflow.
The first central part of this thesis declares three key steps to enable an in-memory event tracing workflow: selection and filtering, encoding and compression, and event reduction. It introduces new enhanced encoding techniques and the novel approach of event reduction that dynamically adapts trace size during runtime to the given memory allocation.
The second central part introduces the Hierarchical Memory Buffer, a new event data representation that uses a hierarchical layout and a highly dynamic distribution of memory bins instead of a single flat continuous memory segment like current event tracing libraries. It allows to perform the aforementioned event reduction operations with minimal overhead.
In the subsequent evaluation, the new enhanced encoding techniques prove to be effective and reduce memory allocation during runtime by a factor of 3.3 to 7.2, while at the same time do not introduce any additional overhead on the tracing library. Furthermore, a case study of the molecular dynamics package Gromacs demonstrates that the concepts of this thesis can remarkably reduce the resulting trace size up to three orders of magnitude, while still providing a coarse but meaningful analysis of the application.

References

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

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

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

[4] Barcelona Supercomputing Center: *Paraver Version 3.0 Tracefile Description*.
http://www.bsc.es/computer-sciences/performance-tools/documentation (last visited 13 Nov 2014)

[5] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. *Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries*. In Applications, Tools and Techniques on the Road to Exascale Computing, Advances in Parallel Computing 22, pages 481–490, 2012.

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

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

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

[9] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf: *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir*. In Tools for High Performance Computing 2011, pages 79–91, Springer, 2012.

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

[11] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel: *Developing Scalable Applications with Vampir, VampirServer and VampirTrace*. In Advances in Parallel Computing 15: Parallel Computing: Architectures, Algorithms and Applications, pages 637–644, 2007.

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

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

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

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