Strategies for Real-Time Event Reduction

Michael Wagner and Wolfgang E. Nagel

Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, 01062 Dresden, Germany
michael.wagner@zih.tu-dresden.de

Abstract. One of the most urgent issues in event tracing is the number of resulting event trace files pushing against the limits of today's parallel file systems. To address this issue, we present strategies for real-time event reduction, which guarantee that data of an event tracing measurement fits into a single memory buffer. Therefore, they are a key step towards a complete in-memory event tracing workflow enabling event trace analysis on very high scales without the limitations of today's parallel file systems. In addition, we define criteria to compare different reduction strategies and evaluate their benefits. Furthermore, we show how traditional memory buffering can be enhanced to realize these strategies with minimal overhead.

Keywords: Event Reduction, In-Memory Event Tracing, Trace Format.

1 Introduction

Development of efficient applications is a challenging task. It demands knowledge of the underlying hardware, compilers and the own source code, which is not trivially given, especially for huge and complex applications. The massive parallelism of today's HPC (High Performance Computing) systems creates another dimension of complexity. Developers have to consider parallel programming paradigms, communication, and load balancing issues to utilize the enormous parallel computing resources. Therefore, the development of efficient parallel software benefits greatly from appropriate supporting tools such as performance analysis tools.

Performance analysis tools collect information about the application behavior and help the developer to better understand his code and to identify performance problems. The two main approaches are profiling and event tracing. While profiling is the gathering of summarized information about different performance metrics, event tracing records runtime events together with a precise time stamp and further event specific metrics. Thus, event tracing collects very detailed information about an application, which reveals a deep insight into its behavior, e.g. communication patterns or load balancing behavior. On the downside, event tracing usually results in huge data volumes since the number of recorded events is quite large. In fact, this is one of the most urgent challenges: the enormous amount of data that is recorded in a single measurement run. Since the recorded

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 429–438, 2013.

data is typically stored in one file per process or thread, especially, the increasing number of resulting files is already pushing against the limits of today's parallel file systems. Without any special treatment, it is possible to handle about ten or twenty thousand of parallel processes. There are already some solutions to circumvent current file system limitations and handle hundreds and thousands of processing elements [2,4]. But, what about future systems with millions or tens of millions of processing elements?

Keeping the data in main memory for the complete workflow from measurement to analysis would completely circumvent file system interactions and, therefore, the file system's limitations. Moreover, it would eliminate the overhead of file creation and writing altogether. In addition, this would enable new ways of event tracing. Our target is to enable *interactive event tracing*. This new workflow would provide faster feedback from the measurement: a measurement could start with a specified set of tracing parameters and interrupt at a defined point. Then, the developer gets a first insight into the application behavior and can decide how to go on: delete unnecessary parts of the measurement, modify the level of detail, or maybe start over because the application is running on a wrong data set. This is just a small glimpse; much more can be done with this approach (see [9]).

But there is one catch: to avoid file interaction, the recorded data has to fit into a single memory buffer. Unfortunately, this is not given for most applications. Quite contrary, a measurement can collect hundreds of megabytes up to gigabytes of data per process. To make things worse, most applications utilize main memory quite well. Thus, the part of the main memory used to store event data should be rather small.

Hence, the challenge is to fit all the data into a single memory buffer. We think of three major steps to achieve this. The first step is a combination of automated intelligent high-level filters, phase based selection and user defined filtering. Currently, we study the effects of runtime loop phase selection, i.e. storing only distinctive representatives of a loop class. There are many different filters and selections to study and we are convinced, that combined they can achieve a very good reduction of "unnecessary" events. The second step is an efficient storage of events in the memory buffer. In [9] we describe our efforts in this area. We were able to increase memory efficiency during runtime by a factor up to 5.8 without increasing the overhead of the event tracing library. In this paper, we focus on the third and last step; a very important one. The two first parts can reduce the memory allocation remarkably but only by a limited factor, i.e. they cannot guarantee that the data fits into a single memory buffer. The third and last part has to ensure that all data fits – always. Therefore, this step is completely different to the filtering and compression steps. It is triggered in the moment 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 challenge is to transparently make space available again by reducing the events stored in the memory buffer. This event reduction must be done with minimal overhead. We call this step *real-time event reduction*.

For a better understanding of the complexity of this last step we must distinguish between high-level information and low-level information. The measurement environment holds a lot of information about the application in its internal data structures. Thus, high-level filters are best located in the measurement environment. The memory buffer management is typically delegated to an event tracing library. This provides many benefits for the event tracing tools like an optimized data compression and interoperability with different analysis tools. But, this also means that at the point where the event reduction step is located (the event trace library), only a subset of low-level information like event classes is available. So, the crux of the event reduction step is to achieve reduction with minimal overhead and minimal information demand.

The contribution of this paper is the presentation and comparison of different new approaches to achieve real-time event reduction and, therefore, guarantee to always fit all recorded data into a single memory buffer. Real-time event reduction is a key step towards a complete in-memory event tracing workflow, which enables event trace analysis on very high scales without today's file system limitations. In addition, we show how these different approaches can be realized. As far as we know, this is novel work and there are no realization strategies presented so far.

The approaches and realization strategies are based on the Open Trace Format 2 (OTF2) [1], a state-of-the-art Open Source event trace library used by the performance analysis tools VAMPIR [6], SCALASCA [3], and TAU [8]. The methods can also be generalized on other event based tracing libraries.

In the following section we distinguish our work from other current approaches. In section 3 we describe the different approaches for real-time event reduction in detail and show realization strategies in section 4. At the end, we summarize the presented work and give an outlook on our future work.

2 Related Work

Since huge numbers of files are an urgent problem in current parallel file systems, a number of potential solutions are in recent development. Two approaches used in event tracing are SIONlib [2] and IOFSL [4]. SIONlib merges multiple logical files into a single or a few physical files. A merged SIONlib file consists of huge pre-allocated segments for the logical file handles; using the file system's capability to handle large sparse files. The I/O Forwarding Scalability Layer (IOFSL) provides an atomic append capability that can be used by applications that have huge parallel per process output like tracing. This allows to write output from many processes into a single or fewer files without any coordination between the processes. Each process also knows the offset where its own data is stored, enabling data access without parsing the entire shared output file.

An alternative is to avoid file interaction at all. While the general idea of keeping all data in main memory for the whole workflow is not a new one, to the best of our knowledge, the strategies of how to do so are not presented so far; except the very basic idea of Section 3.1, which is used i.a. in [3,6].

3 Reduction Strategies

In this section we present three different strategies for real-time event reduction. While these strategies may seem rather simple, we must keep in mind that at the point these strategies operate there is only a limited view on the application's total configuration (see Section 1). In an event tracing library, where the memory buffer is located, there is no information about the call-tree, other processes, or the context of a single event. The strategies need to work with minimal information like the type of an event. Therefore, it is obvious that these strategies cannot be compared with high-level filtering techniques. The main goal of the reduction strategies is to guarantee that there is never an overflow of data in the memory buffer. Or in other words, only a single memory buffer is necessary to record an entire measurement.

The basic idea is to get a coarse overview of the application behavior in any case. Especially at a first measurement the user has only limited knowledge and understanding of the application's behavior. So, user based high-level filtering might be less effective, and, therefore, the main reduction is done with event reduction strategies. The resulting overview of the application's behavior is coarse. But the user gets an overview at all and is able to derive first knowledge about the application behavior. The gained knowledge can then be used to achieve better user based high-level filtering and, with this, a more accurate overview of the application behavior.

Another aspect that needs to be considered is the timing of the reduction. While high-level filtering reduces during the whole runtime, event reduction engages at the moment the buffer is exhausted, i.e. a request to store the next event is issued but there is no space left in the memory buffer. At this moment the reduction strategy intervenes, reduces the amount of already stored events, and makes memory space available again. This reduction must work with very little overhead, so that storing the event is delayed as little as possible and, therefore, the recorded application behavior is minimally disturbed.

These two requirements, minimal overhead and minimal information demand, must be met by all reduction strategies. Thus, these criteria are not feasible to compare the different reduction strategies. For comparison we use the resulting data that is left after reduction. But this can not be evaluated on a quantitative basis. Rather, we must find criteria to review the quality of the remaining information. Performance analysis has two main goals: first, to better understand the application behavior and, second, to identify performance problems. Therefore, our evaluation is based on how good these goals can still be achieved with the reduced data set: is it still possible to understand the application behavior and is it still possible to detect occurring performance problems? Next to that, we analyze the granularity of reduction steps. This means, how much data is discarded in a single reduction step. If the steps are too huge, a lot of data is lost in a single reduction step, even if not necessary.

In the following we present and evaluate three different reduction strategies: reduction by order of occurrence, by event class, and by call stack level.

3.1 Reduction by Order of Occurrence

The simplest strategy is the reduction by order of occurrence, i.e. to stop recording once the memory buffer is exhausted. This provides a complete recorded application behavior from the start until the point recording stopped. After that point there is no information about the application behavior. Thus, a good understanding can be obtained about the recorded part of the application but there is no information about the application behavior after the recording stopped. For the detection of performance problems applies the same condition. A performance problem that occurs in an early part of the application is analyzable without any constraints because the complete event stream of this part is available. Performance problems occurring in a late state of application runtime are not detectable at all since there are no events recorded after the memory buffer is exhausted. Therefore, the quality of the overall information about the application strongly depends on the structure of the application. Since it is not possible to identify beforehand, where performance problems will occur or where the parts are that need better understanding (because this is the outcome of performance analysis), in most cases this strategy delivers poor results. To analyze a specific code region, a program phase selection on a higher level (see Section 1) delivers best results. However, this already implies a good understanding of the application. Nevertheless, the event distribution by occurrence has a very high granularity; an event-wise reduction is possible.

3.2 Reduction by Event Class

The single events that are recorded by event tracing can be categorized in different classes, e.g. entering and leaving a code region, peer-to-peer and global communication, performance metrics like hardware counters, and file interaction. Naturally, not all classes of events are of the same importance to analyze an application's behavior. For example, if a developer wants to analyze the communication behavior, communication events are very important while hardware counters are less important. When looking into single thread performance it is the other way around. Thus, it is possible to order the classes of event types by importance and start reduction with the least important event classes. Since there is no generally meaningful order for all applications, this order should be specified by the user. This strategy can provide a good knowledge about the application behavior represented by the remaining event types. About application behavior deducible by the reduced event types no knowledge can be obtained at all. This also applies for the detection of performance problems. Performance problems that can be recognized by the remaining event classes can be fully analyzed because all events are available. But, performance problems deducible by the discarded event classes can not be detected. In addition, performance problems that only can be derived by a combination of two or more event classes cannot be detected when one of the event classes has been reduced. Hence, the quality of the overall information about the application depends mainly on an appropriate order of event classes by their importance.

Unfortunately our statistical analysis shows that there are only three dominating event classes: enter/leave events, performance counters, and MPI peer-to-peer events (see Figure 1). All other event classes have only a marginal fraction of the total memory allocation. Therefore, this strategy has a limited potential for reduction since the event distribution by event type is coarse. Nevertheless, this strategy can be a good choice to sort out events of one of the main classes if they are of less importance.

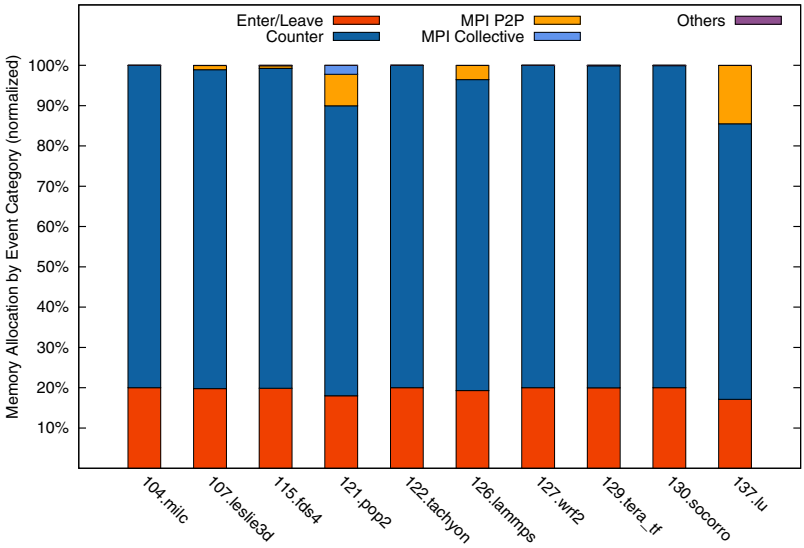

Fig. 1. Memory allocation by event type for SPEC MPI 2007 applications. There are only three dominating event classes: enter/leave events, hardware performance counters, and MPI peer-to-peer events. All other event classes have only a marginal fraction of the total memory allocation.

3.3 Reduction by Call Stack Level

A third strategy is to select events for reduction by their call stack level, i.e. events with the highest call stack level are first to be reduced[1]. This strategy delivers different results according to understanding of application behavior and detection of performance problems than the first two strategies. Of course, detailed information about events of discarded call stack levels is lost, too. But information is still partly available in lower call stack levels. While in the other

[1] Since MPI does not provide any features to determine matching MPI calls (e.g. a receive operation to its according send operation) automatic message matching is an integral part in event trace analysis. But, correct automatic message matching can only be done, if all participating MPI calls are recorded. Therefore, MPI calls are treated separately in this strategy, i.e. either all MPI calls of all levels are kept or all are discarded.

two approaches, for reduced application sections it is not possible to identify a performance problem at all, this strategy provides at least a hint for a potential performance problem. When reducing the call stack level containing events carrying information about a performance problem, the cause of the performance problem is lost but it is still possible to recognize the impact of the performance problem. For example in Figure 2 there is a load imbalance caused by the function *bar* on process two. With a reduction of the call stack level containing *bar*, the cause is not visible anymore. But, the impact in *foo* and, therefore, the performance problem itself is still detectable and can be analyzed in detail with another measurement run focused on this specific code region. Therefore, the knowledge about the application and the ability to detect performance problems is reduced but not completely lost for individual parts.

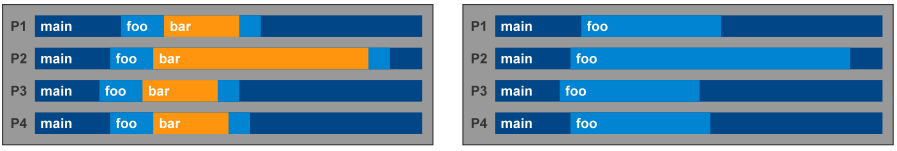

Fig. 2. Timeline representation of a sample load imbalance for four processes. Function *main* is calling *foo*, which is calling *bar*. The load imbalance is caused by function *bar* on process two. Left: the complete event stream. Right: event stream without the highest call stack level.

The applicability of this strategy strongly depends on the application design with regard to call stack level distribution. The best case is a deep and equally distributed call stack (Figure 3 top, left). This enables a reduction in very fine steps. Our statistical analysis showed that there is a variety of different call stack distributions (see Figure 3). There are some applications where a reduction can be done in several steps (Figure 3 top, right and bottom, left) and, therefore, are well suited to this reduction strategy. But, there are also applications where a reduction can only be done in a few big steps (Figure 3 bottom, right). This means, a lot of information is lost within a single reduction step and it is unlikely to utilize the memory buffer well. On the other hand, the statistical analysis showed that the reduction granularity for the analyzed applications mostly is related to the number of events, i.e. applications with a low call stack granularity, typically, produce a moderate number of events. For example, the application 115.fds4 generates an order of magnitude less events than the two other applications in Figure 3.

Altogether, a combination of the different reduction strategies provides best results. For example, when looking into load imbalances it is best to discard the communication events first. After that, it is better to reduce the call stack levels first instead of discarding the performance metric events. Such a combined strategy can utilize the benefits of the single strategies and deliver the most descriptive resulting overview of the application behavior.

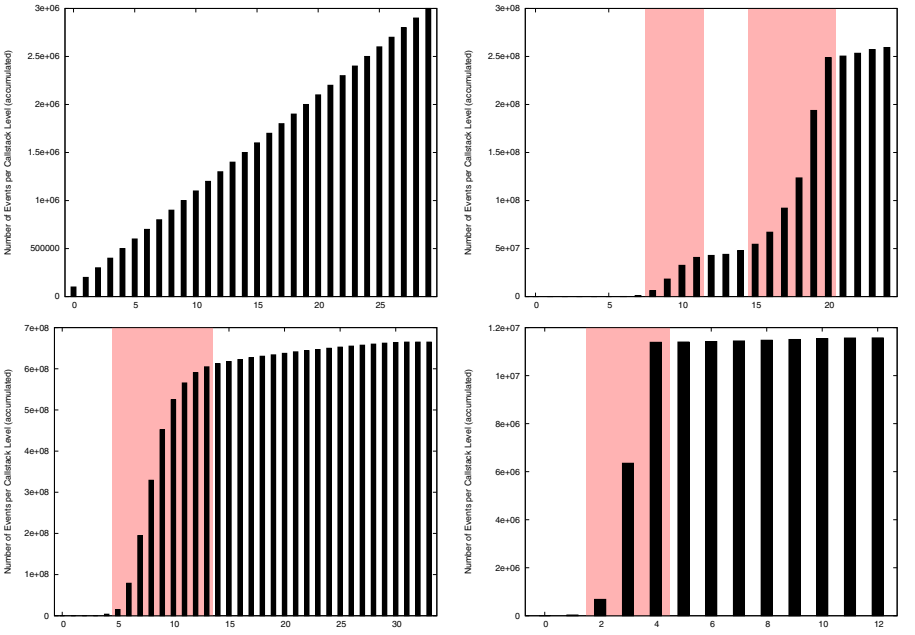

Fig. 3. Event distribution by call stack level (accumulated) for selected SPEC MPI 2007 applications. Top, left: the best case is a deep and equally distributed call stack. Top, right: 130.soccoro - a reduction in 4+6 steps is possible (red zones). Bottom, left: 122.tachyon - a reduction in nine steps is possible. Bottom, right: 115.fds4 - a reduction can only be done in two big steps.

4 Realization Strategies

Typically, event trace formats store the events in the order of their occurrence within a single memory buffer per process [1,5,10]. This is the most trivial form for writing and also the natural order to read the events in again. Therefore, the first reduction strategy that reduces events by the order of their occurrence is easy to apply, e.g. just stop recording once the memory buffer is exhausted. This approach does not require significant modifications to the general event handling and generates basically no overhead.

The two other approaches are much more complex to realize. Using the typical single buffer approach means that the different event types and the events of different call stack levels are scattered over the whole memory buffer. Thus, once the memory buffer is exhausted, all events must be read to identify those who need to be discarded, e.g. those of a specific event class or the highest call stack level. This generates huge overhead and breaks the requirement for a real-time solution. Even more, after the reduction the free memory space is highly fragmented in a lot of very small slots. Hence, this method is not useful at all.

The only way to avoid this situation is to order the events from the beginning by the different reduction criteria. This means, events must be sorted by

occurrence, by event classes, and by call stack level altogether. To do so in a single memory buffer requires a partitioning of the memory buffer in many small fragments reserved for each reduction criteria e.g. one small fragment for each call stack level that might occur or each event class. In the following we use the term *reduction item* to entitle either a single call stack level or a single event class. Each fragment for the different reduction items should be utilized until the total memory usage hits the limit. In addition, after reduction the freed memory space should be available for the remaining reduction items. Obviously, this cannot be done with the classic single memory buffer approach. The distribution of memory to the different call stack levels and event classes must be organized in a highly dynamic way.

To realize this dynamic distribution, instead of one huge buffer, many very small bins (e.g. 64 kB each) are used. These bins are provided on request to the different call stack levels or event classes. Such a request is triggered either when there is no bin available for a reduction item, i.e. the first event for this reduction item should be written or the current bin is full. When there are no bins left and a request is triggered, one reduction item is reduced (e.g. the highest call stack level) and all the bins used by this reduction item are made available for the remaining reduction items; in particular, for the one that triggered the request. To avoid frequent costly allocation and freeing of memory resources a memory management system is applied. Such a memory management system as used in OTF2 by Score-P [7] allocates the complete memory resources at the beginning and delivers small memory pages from the complete memory allocation. With this, the reduction strategies introduce minimal overhead and enable a real-time event reduction.

5 Conclusion

In this paper we presented strategies for real-time event reduction. These strategies guarantee that data of an event tracing measurement fits into a single memory buffer. Therefore, they are a basic step towards a complete in-memory event tracing workflow, which enables event trace analysis on very high scales without the limitations of today's parallel file systems. In addition, we defined criteria to compare these strategies and evaluated their benefits. And, we showed how traditional memory buffering needs to be enhanced to realize these strategies with minimal overhead.

6 Future Work

The presented strategies as well as other optimizations like enhanced compression techniques [9] are based on and developed in cooperation with OTF2. Therefore, the goal is to integrate the strategies for real-time event reduction into the Open Trace Format 2 and make them available to a wide user group. Furthermore, our final goal is the realization of interactive event tracing as described before to enable faster insight into the application behavior. The implementation

of these strategies enables in-memory event trace analysis and, therefore, is a key step towards interactive event trace analysis.

References

1. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2 – The Next Generation of Scalable Trace Formats and Support Libraries. In: Proc. of the International Conference on Parallel Computing, ParCo, Ghent, Belgium (2011) (accepted for publication)
2. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel i/o to task-local files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 17:1–17:11. ACM, New York (2009), http://doi.acm.org/10.1145/1654059.1654077
3. Geimer, M., Wolf, F., Wylie, B.J., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca Performance Toolset Architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
4. Ilsche, T., Schuchart, J., Cope, J., Kimpe, D., Jones, T., Knüpfer, A., Iskra, K., Ross, R., Nagel, W.E., Poole, S.: Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In: Proceedings of the 21th International Symposium on High Performance Distributed Computing, HPDC 2012. ACM (June 2012)
5. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
6. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool Set. In: Tools for High Performance Computing, pp. 139–155. Springer (July 2008)
7. Knüpfer, A., Rössel, C., an Mey, D., Biersdorf, S., Diethelm, K., Eschweiler, D., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Tools for High Performance Computing 2011, Dresden, Germany (2011) (accepted for publication)
8. Shende, S., Malony, A.D.: The TAU Parallel Performance System, SAGE Publications. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
9. Wagner, M., Knüpfer, A., Nagel, W.E.: Enhanced Encoding Techniques for the Open Trace Format 2. Procedia Computer Science 9, 1979–1987 (2012)
10. Wolf, F., Mohr, B.: EPILOG Binary Trace-Data Format. Tech. Rep. FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich (May 2004)