Enhanced Encoding Techniques for the Open Trace Format 2

Michael Wagner, Andreas Knüpfer, Wolfgang E. Nagel

Abstract

Highly efficient encoding of event trace data is a quality feature of any event trace format. It not only enables measurements of long running applications but also reduces bias caused by intermediate memory buffer flushes. In this paper we present encoding techniques that will remarkably increase memory efficiency without introducing overhead for the compression. We applied these techniques to the Open Trace Format 2, a state-of-the-art Open Source event trace data format and library used by the performance analysis tools Vampir, Scalasca, and Tau. In addition, we show that these encoding techniques are a basic step in achieving a complete in-memory event trace workflow.

Keywords: Encoding Techniques, Compression, Event Tracing, Trace Format, OTF2

1. Introduction

Performance analysis is the basis for effective performance optimization, especially for large parallel applications. Therefore, performance analysis tools for parallel applications form a vital part of the HPC software landscape. Of that, event trace recording and post-mortem analysis is one of the two main branches; profiling is the other. The basic idea of event tracing is to log runtime events (e.g. entering or leaving a function) together with a precise time stamp and further event specific information. This allows a detailed post-mortem analysis of the parallel behavior and in particular the timing, which is the most prominent criterion for computing performance [1, 2, 3]. Besides pure contemplation of recorded events, the same method can be used as basis for detailed event replay or performance prediction [4, 5].

The nature of event tracing is to collect and store very detailed information. Even though single event records that describe single events are rather small, this frequently results in huge generated data volumes, because the number of events is quite large. In all cases, a low overhead during the trace generation process is desirable – either in terms of run-time alteration or in terms of memory consumption. For the purpose of performance analysis it is most vital because it will reduce the disturbance of the behavior under observation. One of the components involved is the event trace data format. It controls the in-memory buffer during the recording period as well as the trace file format when data is written to permanent storage at the end of a recording period. It is responsible for overhead in terms of timing and memory consumption.

Up to now, trace compression is done only to minimize storage space for traces on the file system. External compression libraries are very strong with regard to memory efficiency as well as time efficiency. Nevertheless, time for data compression is too high in relation to time granularity of event tracing. Therefore, trace compression, so far, is only applied after the measurement has finished. Thus, the main challenge in reducing memory consumption during
run-time is to increase memory efficiency without decreasing time efficiency. This is impossible with any compression library. The contribution of this paper is a set of encoding techniques that meet this challenge: remarkably increase memory efficiency without introducing overhead for the compression. These techniques are based on and applied experimentally to the Open Trace Format 2, a state-of-the-art Open Source event trace library used by the performance analysis tools Vampir, Scalasca, and Tau. Therefore, a wide group of users will benefit from these improvements. In addition, these techniques can be generalized and applied to other trace formats with a similar basic design.

In the following section we give an overview over existing event trace formats and their design goals. After that, in section 3, we describe the different encoding techniques we applied. The results of these techniques are evaluated in section 4. At the end, we explain our future research goals based on this results in section 5 and, finally, summarize the paper.

2. Related Work

Event trace data formats used to be seen as an integral part of the different performance tools only. They have been considered as a mere transport layer moving information from the run-time measurement to the post-mortem analysis. Also, all tools used to implement their own formats with different APIs and slightly different terminologies, even though the fundamental workings and the pieces of information covered were almost the same.

This was true or still is true for the following existing formats. The CLOG and SLOG family of formats were an integral part of the Jumpshot families of viewers [6, 7, 8]. The format contains almost ready-made graphical representations that can be loaded selectively to visualize a selected part of the data set on screen. To the best of our knowledge, there is no active development behind SLOG and Jumpshot anymore. The Epilog Trace format is the native format that comes integrated with the Scalasca tool-set. It strictly uses one output file per process/thread and is tailored towards the replay-based automatic search for well-understood typical performance problems by Scalasca [9]. The Paraver Trace Format is the native format of the Paraver trace visualizer and the Dimemas simulator. Paraver’s format is more generic than all other formats: instead of predefined data-records for all typical events it uses generic data tuples that can be interpreted in different ways. There exist more trace formats in the HPC performance analysis areas. Also, there is some degree of interoperability via format converters, even though converting huge data sets is increasingly problematic. A general overview about trace formats can be found in [10].

The first trace format developed with interoperability in mind was the Open Trace Format (OTF) [11], created by Technische Universität Dresden (Germany), University of Oregon (USA) and the Lawrence Livermore National Lab (USA). It consists of a defined API together with a read/write library and a number of support tools. As the first format it was distributed as a separate software package and was adopted by the Vampir tool-set, the TAU tool-set, and the Microsoft HPC Server built-in MPI trace collection layer, among others.

The successor version OTF2 was jointly developed by Technische Universität Dresden (Germany), the Jülich Supercomputing Centre (Germany), the German Research School for Simulation Sciences Aachen (Germany), Technische Universität München (Germany), University of Oregon (USA), RWTH Aachen University (Germany), and GNS GmbH Braunschweig (Germany). It is part of Score-P, a software infrastructure for instrumentation and run-time recording, and will become the common native event trace format for Vampir, Scalasca, and TAU [12, 13]. There have been many successive improvements in the way from early simple event trace formats to the tool-specific formats to OTF and OTF2. For example, OTF introduced selective access to the trace data in the time and space dimensions. OTF2 adapted the built-in buffering layer so it can be re-used as the in-memory record collection buffer in the run-time measurement system [12].

Many event trace formats employ general-purpose data compression. Besides that, there are also special-purpose semantic compression techniques for event trace data. An early approach was presented in [14]: conflating trace data streams from separate processes or threads, if they were similar enough. Another approach is the CCG data structure [15, 10] which captures similar reoccurring event sequences in a trace, within or across different processes and threads. It keeps detailed timing information and allows controlling bounds of timing deviation due to the semantic compression. It is not intended as a general file format but as a analysis-time memory data structure. The Scalatrace semantic trace compression [4] uses extended regular expressions to represent repeated event patterns within or across different processes/threads. It was designed as a file data format, yet for the purpose of event replay. Therefore, it keeps only coarse timing histograms, which are insufficient for post-mortem performance analysis. These semantic
based compression techniques are all very complex and costly and do not provide an alternative to the techniques in this paper. To the best of our knowledge, there are no event trace formats that use efficient on-line trace compression, so far.

3. Encoding Techniques

In this section we will present the different encoding improvements that noticeably increase memory efficiency of the trace format. The first two optimizations are already implemented in the release version of OTF2: the leading zero elimination and the separation of timing information and event data. Both were already presented in [12]. Hence, they are the basis for all further encoding techniques and will be described shortly in the following as well. As far as we know, all other presented techniques form a novel approach in on-line trace compression. They were neither presented nor implemented in any format, yet. All measurements in Section 4 present the advantages of the new techniques in comparison to OTF2, which already contains the first two techniques.

But first, we want to explain the basic memory representation of event records, e.g. in OTF. 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, etc.); second, a time stamp telling when the event occurred; and third, event specific attributes, e.g. a region ID for an region enter record (see Figure 1).

![Figure 1: Basic memory representation of event records: first a one-byte record token that defines the type of an event (e.g. entering or leaving a code region, sending or receiving a message, etc.); followed by a time stamp of eight bytes telling when the event occurred; and third, event specific attributes (e.g. a region ID for an region enter record) with four bytes each.](image)

3.1. Splitting of Timing Information and Event Data

Most trace formats store events and their according timing information in a single event record (see Figure 1). This is consequential to the general event tracing approach because an event is only meaningful with information about the time it occurred. However, the advantage of storing timing and event data separately (see Figure 2) is the elimination of redundant timing information for consecutive events with same time stamp, which occurs quite often. For example, calling MPI_Send() results in the following event pattern: an enter event for the MPI_Send region, an MPI send event with the message-specific information, and an leave event for the MPI_Send region. The timing information of the MPI send event is matched either to the enter or the leave event. This means, for this example, there are only two time stamps stored instead of three, i.e. a reduction of 8 bytes. The trade off is an additional byte for a token for the new time stamp record.

![Figure 2: Splitting of timing information and event data results in two separate records: first a record storing only the time stamp that is valid until the next time stamp record; and second, a record storing only the event attributes.](image)

3.2. Leading Zero Elimination

The memory reserved for single attributes of an event is usually determined by the largest value this element has to represent. Therefore, a region ID is typically stored in a 32-bit integer while a performance counter is stored in a 64-bit integer. However, most values are much smaller than the hypothetical maximum but result in the same memory
allocation. By omitting bytes that are zero, memory allocation for most values is reduced. To still be able to read this not fix-length representation of the value the number of remaining data bytes has to be stored in front of the value. Figure 3 shows this for a simple event record with a four-byte and an eight-byte attribute.

![Figure 3: Leading zero elimination: all bytes that are zero are omitted. The number of the remaining data bytes its stored in front of them.](image)

### 3.3. Difference Storing

Basically, there are two different types of values that need to be stored: monotonic increasing values like time stamps and arbitrary values like region IDs. Furthermore, some of the monotonic increasing values begin with a very high offset. Storing only the difference to the previous value leads to much smaller values to store. In combination with the described leading zero elimination this results in less memory allocation for the stored value. At the moment this is only applied to time stamps. To still allow random access to events, the buffer is divided in several small blocks. Each block begins with a complete time stamp. This way, access to events is not completely random but it is not necessary to read the whole trace from the beginning.

### 3.4. Bit-Level Encoding

The previous optimizations can be optimized even more by encoding very small numbers directly into the token byte. With the leading zero elimination (see Section 3.2) a token is always followed by a very small number: the number of remaining data bytes. This number can be easily encoded in the token byte by simply adding the value to the token (see Figure 4). This eliminates the additional byte storing the number of data bytes. Of course, we had to consider the addition in the distribution of token IDs to still enable an unique identification of events. Because the token byte is limited to 256 values, only the most frequently occurring events (see Figure 5) use this encoding: time stamps and enter/leave records. To make decoding easier there are token for each record length: time stamp with one data byte, time stamp with two data bytes, ..., time stamp with eight data bytes. With these two optimizations (leading zero elimination and bit-level encoding) the size of the sample record is reduced from thirteen to seven bytes.

![Figure 4: Merging of token and length byte: the number of remaining data bytes of the first attribute is encoded in the token byte by simply adding the value to the token.](image)

### 3.5. Event Distribution and Encoding Implications

So far, the presented encoding techniques are a benefit for most event records with a reduction of memory allocation. But to reduce the overall memory allocation even more, it was necessary to identify those event records that consume a lot of memory in total and, therefore, are preferential for further optimization. Figure 5 shows the results of our survey: the distribution of the different event record types is far from uniform. Enter and leave events
are the dominating events in typical application traces. In addition, most traces contain only a moderate number of program regions. This means the region ID parameter in enter and leave event records is moderately small. Therefore, these two events bear a high potential for further memory reduction. As a result of this statistical survey enter and leave event records are treated differently from other event records. As mentioned before, enter records are already optimized by bit-level encoding. If the value of an enter’s region ID is small enough the value is directly encoded in the event token (see Figure 6). Otherwise only the length of the following data bytes, as described before, is encoded in the event token. This means that for many application traces the size of most enter records is reduced to only one single byte. For leave records the region ID is not stored anymore but can be delivered by keeping track of the function call stack. Thus, leave records always require only a single byte. That means, that the size of most enter and all leave records is reduced from five bytes to a single byte.

Figure 5: Memory allocation by event type for SPEC MPI 2007 applications: Enter and leave events are the dominating events.

Figure 6: Merging of token and value: if the value of an enter’s region ID is small enough the value is directly encoded in the event token

---

1The 256 values in the token byte are distributed among three domains: token for events, reserved token for upcoming events, and the remaining token for directly encoding enter events including region ID.
3.6. Timer Resolution Reduction

Typically, tracing tools check for available timer sources and use the one with the highest resolution e.g. a CPU cycle counter. However, events occur in a much lower frequency. In addition, the remaining error after post-mortem synchronization of non-global timer sources is usually in the range of microseconds [16]. Therefore, it is possible to reduce the stored timer resolution of high frequent timers without perceivably degrading the timing information. The reduced timer resolution then is still a few orders higher than the event frequency. This ensures that derived values (e.g. total execution time of a function) can still be computed correctly. An optimal reduction factor\(^2\) can be determined after the measurement with less effort but it is very difficult to do it in before. This is still work in progress. Thus, we use only a very slight reduction for high resolution timers from our experience. No reduction is done for medium and small resolution timers to ensure the program behavior is not modified.

4. Evaluation

Most significant in evaluating an event tracing library besides scalability and usability, are memory consumption and the introduced runtime overhead. The presented encoding enhancements will only affect the internal memory representation. Thus, they will not lead to an impact on OTF2’s usability or scalability. However, they will influence the consumed memory and runtime overhead. In the following we will show that the optimizations will drastically reduce the memory allocation without increasing the runtime overhead. Therefore, we compared OTF2 with and without the encoding techniques as well as its predecessor OTF.

However, comparing different event trace formats is not a trivial task. In most cases, event tracing libraries are very closely integrated in the according event tracing tools which generate their own individual overhead. Also, they generate different values for events, e.g. time stamps. On the other hand, generating events with random or fixed values is not an option because compression ratio strongly relates to the type of input data and, therefore, would lead to misleading results. Thus, we developed a simulator that reads in an existing trace file and writes the records to the different event trace formats. With this, all three event traces contain exactly the same real-life data.

As reference event traces we used a subset of the SPEC MPI 2007 benchmarks\(^3\) that represent a variety of applications from different research fields, different communication behaviors and different length, i.e. trace sizes.

4.1. Runtime Memory Consumption

The primary goal of the encoding enhancements is to drastically increase memory efficiency, which leads to less interrupts for storing collected data to disk and, therefore, more accurate results. Next to that, the overall storage size of an experiment drops to a much smaller amount. Figure 7 shows the effects of the presented encoding techniques. To be able to classify the capability of these enhancements we compared them to the memory efficiency achieved with zlib\(^4\) - a well established general purpose compression library. The figure shows the results for OTF, OTF2, OTF2 with zlib compression, OTF2 only with the lossless enhancements (ee), and OTF2 additionally with the lossy timer resolution reduction (rtr). The results show that the encoding enhancements reduce the memory consumption by a factor of 3.6 to 5.8 and, therefore, realize memory efficiency equal to or better than a compression library. An additional nice effect is that the compression ratio with zlib of trace data with enhanced encoding is 5.4 to 9.0. This means, it is still notably higher than without enhanced encoding which compresses with a ratio of 3.5 to 4.7.

4.2. Runtime Overhead

Next to the memory efficiency, time efficiency is equally important for a successful measurement of an application. Introducing too much overhead will drastically reduce the accuracy of a measurement or might even record a different behavior than the real application. This means, an event trace format as part of an event tracing tool has to introduce as less overhead as possible. Therefore, improvements in memory efficiency must be judged by the overhead they introduce.

\(^2\)An optimal reduction factor is the highest possible reduction before the program behavior is modified in any way i.e. events that occur at different original time stamps occur at the same reduced time stamp.

\(^3\)http://www.spec.org/mpi2007

\(^4\)http://zlib.net/
Figure 7: Memory consumption of OTF without zlib compression, OTF2 without zlib compression, OTF2 with zlib compression, OTF2 only with the lossless enhancements (ee), and OTF2 additionally with the lossy reduced timer resolution (rtr). All values are normalized to OTF’s memory consumption.

Table 1 shows on the left side the time consumption of OTF, OTF2, OTF2 with zlib compression, and OTF2 with enhanced encoding. It shows two important results: First, the enhanced encoding does not introduce new overhead. The overhead is in the same range as OTF2 without the enhanced encoding. Second, using a compression library to increase memory efficiency will drastically decrease time efficiency. Even more, overhead introduced by a compression library is usually not equally distributed over all events. In most cases, a certain amount of data is collected and then compressed in one step to achieve a high compression ratio. In our measurements, compressing every 1 MB creates an overhead of approx. 24 ms in average. Therefore, using a compression library for higher memory efficiency is unfavorable.

Table 1: Time and memory consumptions of OTF, OTF2, OTF2 with zlib, and OTF2 with enhanced encoding (including timer resolution reduction) for selected SPEC MPI 2007 test cases. All values are normalized to OTF’s time or memory consumption, respectively.

<table>
<thead>
<tr>
<th>Trace Format</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTF</td>
<td>100.0 %</td>
<td>100.0 %</td>
<td>100.0 %</td>
<td>100.0 %</td>
<td>100.0 %</td>
<td>100.0 %</td>
</tr>
<tr>
<td>OTF2</td>
<td>52.8 %</td>
<td>54.1 %</td>
<td>55.2 %</td>
<td>75.9 %</td>
<td>80.6 %</td>
<td>85.1 %</td>
</tr>
<tr>
<td>OTF2 with zlib</td>
<td>231.4 %</td>
<td>249.9 %</td>
<td>285.5 %</td>
<td>17.7 %</td>
<td>19.5 %</td>
<td>23.1 %</td>
</tr>
<tr>
<td>OTF2 with enhanced encoding</td>
<td>50.9 %</td>
<td>51.5 %</td>
<td>52.4 %</td>
<td>13.8 %</td>
<td>17.8 %</td>
<td>23.8 %</td>
</tr>
</tbody>
</table>

With this, the goal to drastically increase memory efficiency without increasing runtime overhead was clearly achieved.
5. Future Work

One of the most urgent challenges in event tracing is still the massive amount of data collected during a tracing run. And, this amount of data is steadily growing in three dimensions: first, by targeting even more detailed information; second, for even longer application runs; and third, on even bigger number of cores. Especially the ladder is already pushing against the limits of today’s – and sadly, most likely also tomorrow’s – parallel file systems. Without any special handling, current tracing tools are capable to handle about ten or twenty thousand of parallel processes. There are some first steps to overcome today’s limits and push them to hundreds and thousands processes [17, 18]. But what about millions or tens of millions of cores?

Next to that, current data volumes are already way to big to be analyzed on single or a few processes. So mostly, event trace analysis is already realized in a highly parallel distributed fashion [2]. Some tools like Scalasca [3] use even the same number of cores for measurement and analysis. However, the typical workflow still is, to record an application’s behavior, store it to disk, and then read it in again to analyze it. Obviously, it seems a good idea to just keep the data in main memory for the complete workflow. This will not only provide advantages in file system interaction but will reduce total time to insight. In addition, it will enable new ways of event tracing. Our final goal is to enable interactive event tracing. Only some potential features could be to start an application measurement with a set of tracing parameters, e.g. number of counters, level of detail, etc., then stop at a defined point, look into the first part of the measurement, get first insights, and then decide how to go on: delete unnecessary parts from the recording buffer, set a new level of detail, choose different counters, start over, abort, etc. This should only give a first impression on the possibilities this new approach might offer.

But, there is one catch: keeping the data in main memory the whole time implies that always all data fits into a single memory buffer. Unfortunately, this is not the case today. Quite contrary, they produce hundreds of megabytes to tens of gigabytes of data per process. In addition, a tracing buffer should only use a small part of total main memory because typically the measured application needs most of it. Therefore, data is not only stored to file at the end of each measurement run, but every time a memory buffer is exhausted, which can be quite often.

That is why, it is absolutely essential to fit all data into a single memory buffer. In our opinion there are three key parts in achieving this goal: intelligent high-level filters and phase-based selection form the first part. So, only valuable events are stored. Of course, this is not as trivial as it sounds. Currently, we study the effects of runtime loop phase selection by classifying loop phases and storing only a representative of each loop class. There is a wide field of different filters and selections to research, which can achieve very good results, when combined together. The second is a very efficient storage of events. This topic is addressed in this paper but we think there is still more potential. In contrast to this two parts that reduce the amount of data by a factor, the last part has to reduce whatever amount of data is left to a fixed size – the size of the tracing buffer. This last step is a very important one because once this step is realized, it guarantees that any reasonable measurement will fit into a single tracing buffer – and this, of course, with very little overhead. This step, called real-time event reduction, is our current subject of research.

Therefore, the presented encoding enhancements provide a great advancement for today’s event tracing tools, yet, they are only one step on the path to a complete in-memory event tracing workflow, which is the basis for an interactive event tracing approach.

6. Conclusion

In this paper, we presented novel techniques for an enhanced encoding in the Open Trace Format 2. By applying these techniques we reduced the memory allocation for event trace data during measurement runtime by a factor up to 5.8 without increasing the overhead of the tracing library. With this, we achieved the memory efficiency of well-established compression libraries without their negative impact on the recorded application behavior. Furthermore, we presented our future goal in providing a complete in-memory event tracing workflow that enables the user to get results faster and less prone to bias by intermediate memory buffer flushes. In addition, it is the basis to evolve from post-mortem event trace analysis to interactive event trace analysis.


