Introducing the Open Trace Format (OTF)

Andreas Knüpfer, Ronny Brendel, Holger Brunst,
Hartmut Mix, and Wolfgang E. Nagel

ZIH, TU Dresden, Germany
{andreas.knuepfer, ronny.brendel, holger.brunst
hartmut.mix, wolfgang.nagel}@tu-dresden.de

Abstract. This paper introduces the new Open Trace Format. The first part provides a small overview about Trace Format Libraries in general and existing Formats/Libraries and their features. After that the important requirements are discussed. In particular it concerns efficient parallel and selective access to trace data. The following part presents design decisions and features of OTF comprehensively. Finally, there is some early evaluation of OTF. It features comparison of storage size for several examples as well as sequential and parallel I/O benchmarks. At the end, a conclusion will summarize the results and give some outlook.

1 Introduction

Today, there are many established trace formats already. Let's name for example Epilog [11], Pajé's trace format [3], Pablo Self-Defining Data Format (SDDF) [1], Structured Trace Format (STF) [4], TAU trace format [2, 10] and Vampir Trace Format 3 (VTF3) [8]. All of them share the basic principle storing event information in so called *event records* which are sorted by time. Special definition records deliver auxiliary information and allow more efficient storage. Event types usually supported by all formats are: function call events, point to point message events, collective communication events, performance counter samples and few more. Usually, there are only minor differences in representation. The general functionality is the same.

To all *trace formats* there is a *trace format library* as well[1]. It serves as read/write layer and passes trace records from/to files. Although all formats could be written or read directly with more or less effort it is not recommended. Instead, one should always use the supplied library. It will perform parameter checking, parsing, data filtering etc. Most of the time this will be easier, safer, faster and more complete.

In High Performance Computing (HPC) there are some special requirements for trace formats and their support libraries. Access speed and storage efficiency are important. Furthermore, both properties must scale well with very huge amounts of data. Parallel I/O is one necessary concept to accomplish this on today's HPC platforms. To incorporate parallel I/O efficiently the trace format library must provide special support.

It turns out, that even the trace formats are affected. Only with data management and storage scheme adapted towards parallel I/O maximum performance becomes achievable. This feature is missing from all but one of the trace file formats mentioned above.

[1] The terms "trace format" and "trace format library" are often used synonymously.

STF [4] implements parallel I/O and advanced selective access. Unfortunately, that format is not available on all platforms and for all purposes (anymore).

This paper presents the OTF format itself and especially the trace format library. It is not intended as comprehensive programming documentation but an overview of functionality. In the first part of the paper there will be a definition of requirements. It is followed by a presentation of the design considerations and main features. Finally, there are some (early) performace data as well as comparisons with the state of the art.

2 Requirements to a Modern Trace Format

First of all, a complete set of record types for the usual purposes has to be provided. The interface should use the same basic form as all existing trace format libraries: It provides record type specific write functions and a call-back mechanism for reading.

The format must be platform independent, especially with respect to endianness and word size (32bit/64bit). Of course, economical storage size is very important. Last but not least, read/write access needs to be fast and efficient. This involves both, selective access and sequential linear read/write operations. The former is the key to parallel I/O which relies on selecting disjoint parts of a trace to be read independently. The latter is relevant for sequential and parallel performance as it determines sustained I/O speed.

Selective access is to be provided with respect to three criteria: First, selection per record type as supported by all existing trace formats. This simply means to ignore unwanted record types. Second, selection per trace process. This must not read everything dropping unwanted parts because it would overstress I/O. Loading only worthwhile data saves a lot of effort. In case a single process contains still too much data the third selection facility can reduce this further. It allows to select a certain time interval (per process). Again, it is important that only the requested data is loaded rather than the whole process trace.

The last part of selective access requires another new facility. In order to start reading at an arbitrary time stamp additional helper information is necessary. Otherwise, the initial state at that time stamp would be unknown, rendering all following events useless. Therefore, a special class of records has to provide resumption points, i.e. full state information at selectable time stamps.

In order to provide a clue what processes resp. time intervals to select *before* reading the whole trace there should be another kind of auxiliary information. This is to provide statistical overview information about what happens on certain processes and in certain time intervals. Those information should be small and concise and yet allow arbitrary granularity.

3 Design Decisions and Features

Based on above requirements the following basic design was chosen: OTF uses *multiple streams per trace*. It uses an *ASCII encoding* which allows *binary search on files* for time intervals. Furthermore, it supports *optional auxiliary information*. Please, see below for a comprehensive description.

3.1 Multiple Streams

The most important point regarding parallel I/O is the distribution of a single trace to multiple *streams*. Those streams are separate files that can be accessed independently. A stream contains events of one or more processes in temporal order. But every trace process/thread is mapped to one stream exclusively, i.e. a process cannot span over multiple streams. This is necessary to maintain the original order of events with identical time stamp in one process. Thus, a trace of n processes can be stored with $1 \dots n$ streams.

In addition, there is an index file with suffix .otf that stores the mapping of processes to streams. It can be set or queried via the trace format library, too. During tracing it might be very convenient to use n independent streams. Simply adding the index file as well as a global definitions file will produce a perfectly valid trace without explicit merging[2]. Later, it might be convenient to reduce the number of streams. Perhaps several thousands of files are undesirable for massive parallel traces. The **`otfmerge`** tool is able to translate this to any number $1, \dots, n$ of streams or to a fixed number of processes per stream.

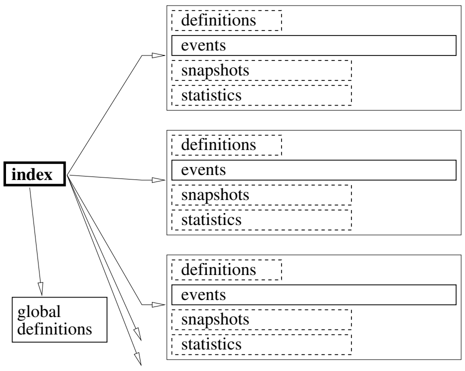


Fig. 1. OTF storage scheme. Index file, global definitions file as well as event files are mandatory. All local definitions, snapshots and statistics are optional.

When writing a trace, the process to stream mapping can either be specified explicitly or created automatically. With respect to reading event records the process-to-stream mapping is completely transparent. The user specifies which processes are to be read. Then, the OTF library accesses only streams containing those processes. If multiple streams are read they are merged on-the-fly to provide a single sorted input stream to the caller. Furthermore, an arbitrary number of streams can be handled with limited numbers of file handles. Figure 1 gives an impression of OTF's storage scheme.

3.2 ASCII Encoding

The internal representation of records in OTF uses an ASCII encoding. This makes OTF platform independent in the purest way. Of course, this is a slight disadvantage in terms of storage size which is compensated later. This encoding makes OTF human readable and accessible via tools like grep, awk, perl, etc.

With this ASCII representation every record is stored as a single line of text. Information about time stamp and process of event records is separated from the records

[2] Merging and post-processing of trace data are frequently used for several purposes. It might still be necessary for timer-synchronization, token renaming, etc.

and managed by a *state machine* where every stream keeps track of current time and process. Special records update those values which then remain until the next update. Ordinary event records contain no time stamp or process but simply refer to the current state of the stream. By this means redundant time stamp and process specifications are eliminated within streams.

Another advantage of the ASCII representation is the ability to recover reading at arbitrary file positions. After a **`fseek()`** operation to any file position a simple forward search for the next newline character will produce a valid resumption point. Further search will deliver an update of the stream's state machine such that reading can be started at that point in a reliable way. This is most important for selective access as introduced below. Unlike for pure binary formats there is no need for anchor information, escape sequences, padding, etc.

3.3 Efficient Selective Access

OTF provides selective access to traces according to three aspects: by record types, by processes and by time intervals.

Selection by record types is possible by disabling a certain record type's call-back handler. The OTF library will then skip all records of that type. It will even avoid fully parsing such records.

For selective reading by processes the user requests a set of processes to be read. This may range from a single process to all. Based on that, OTF accesses only affected streams. If a stream contains enabled as well as disabled processes, it is read and filtered. Thus, only records of the enabled process(es) are passed to the consumer. Again, records of disabled processes are skipped without parsing.

The OTF interface allows only to select by processes but not by streams. However, the user can query the process-to-stream mapping and derive an optimal selection. This is highly recommended for parallel reading.

By this means, multiple processes or threads of a parallel application can read/write disjoint parts of a trace concurrently. OTF does not do any parallelism internally but adapts to the parallel paradigm of the user application. The OTF library is thread-safe but doesn't contain any explicit multi-threading or multi-process parallelism internally. User applications need to take care of race-conditions during parallel OTF output!

Finally, OTF also supports selective access with respect to arbitrary time intervals. From the user interface point of view this works by simply restricting the time interval to be read. Internally, OTF performs a binary search for the start time stamp on all according files. With $O(\log n)$ effort for total record count n this scales very well for large files. Once the start position is found the OTF library performs standard reading until the end of the time interval is reached or all streams are exceeded.

3.4 ZLib Support

For the sake of efficient storage size OTF supports transparent ZLib [6] data compression as an experimental feature. This will compensate for the ASCII encoding which is not most economical by itself.

Every file that is part of an OTF trace can be compressed by itself, it is automatically detected when reading. OTF uses a rather small compression level which allows fast compression and yet good data reduction to 20% - 35% of the original size on average.

ZLib compression is achievable directly during trace file creation or later via the **`otfcompress`** tool, which also supports de-compression. In order to support random access even for compressed files OTF uses a blockwise compression scheme as directly supported by ZLib. Default block size is $10 \ mathrmKB$ which allows good compression and fine-grained access at the same time.

So far, this paper mentioned event records only. Of course, there are definition records, too. They specify certain global properties, like timer resolution and token definitions. Such global definitions are never performance critical and are stored in a single global file (see Figure 1).

OTF supports local definitions for convenience. They are attached to streams but not to individual processes inside a stream. This is provided for trace libraries which might want to store local definitions at trace time. However, local definitions should be translated into preferred uniform global definitions.

Besides definition records there are two more classes of auxiliary records. First, there are so called *snapshot records* which assist user applications when reading arbitrary time intervals. They provide information about the complete state of a trace process at a certain time. This includes the function call stack, pending messages, etc.

Second, so called *summary records* allow a reasonable guess about which processes resp. time intervals to chose for selective access. They provide a variety of properties for certain event record types. For example number of calls and exclusive/inclusive runtime per function. Alternatively, it can be summarized for groups of functions instead of single functions.

Summary information at a time stamp t covers the time interval $[0, t)$, i.e. from the beginning of the trace to the current time. All properties S can be derived for the interval $[a, b)$ by the simple transformation

$$S\big([a, b)\big) = S\big([0, b)\big) - S\big([0, a)\big). \tag{1}$$

By this scheme of representation only n summary points are sufficient to cover $n \cdot (n-1)$ intervals of time.

Both, snapshot and summary records are optional and reside in separate files per stream (see Figure 1). This allows to generate, re-create or remove them without changing the event information. Snapshots and summaries can be supplied by a trace library, by any user application or by the **`otfaux`** tool which adds both to an existing trace with selectable granularity.

4 Early Evaluation

Below early performance evaluation is shown. It was performed on a commodity workstation (AMD Athlon 64 3200+, 2GB main memory) and on a brand-new super computer (SGI Altix 3700, 128x Intel Itanium2 1500MHz, 490GB shared memory, very fast I/O).

Storage Size. The first interesting issue concerns storage size which is a platform independent issue. Fig. 2 compares storage size for identical traces in STF [4] and VTF3 [8] formats to plain and compressed OTF. The examples are original STF traces with 18 MB, with 1.8 GB and with 2.3 GB size from well known ASCI benchmarks [5, 9].

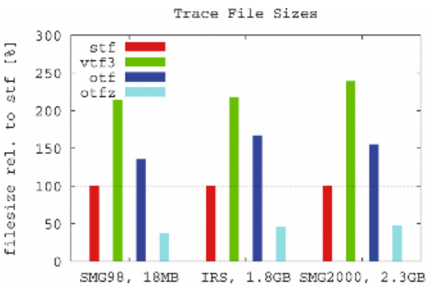


Fig. 2. Storage sizes for STF and VTF3 vs. plain and compressed OTF

Both, compressed and even plain OTF traces are much smaller than binary VTF3, i.e. even a pure ASCII encoding results in a quite efficient storage scheme. Yet, plain OTF is 30% to 60% larger than STF. However, ZLib compressed OTF format turns out to use $< 50\%$ of STF's storage volume. Thus, in terms of storage size compressed OTF provides a notable advantage over two well established binary trace formats.

Read Performance. The second issue of interest is raw read performance of the trace format libraries. Traces are read, parsed and send to dummy handlers measuring the formats' read performance including internal overhead.

Figure 3 shows reading speed in million records per second for the 3 example traces in the 4 traces formats. On the AMD 64 platform (Figure 3 left) there is an obvious difference between small traces (18 MB), where I/O is noncritical, and huge traces ($>$ 1 GB), where I/O becomes the limiting factor.

For the small example uncompressed OTF speed is almost double of STF despite the larger storage size. Reading compressed OTF is no advantage here. Instead, the additional computation decreases performance while reduced input amount ($\approx 1/3$) brings only minor benefit.

For large traces the situation is different. Now, the run-time behavior looks perfectly reciprocal to the storage size diagrams. The I/O effort determines the read performance and thus, compressed OTF is fastest by far.

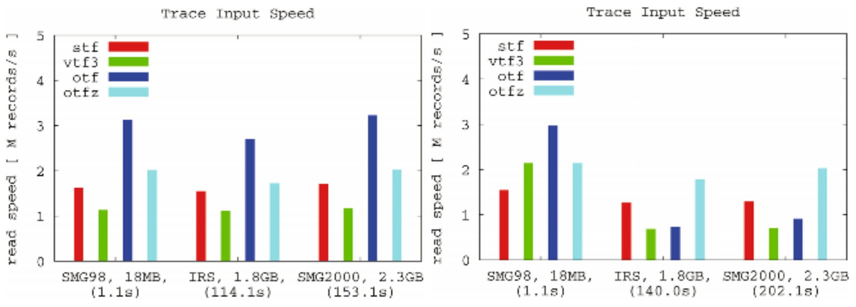
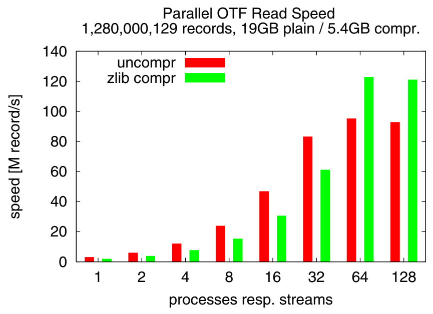


Fig. 3. Read speed for three real-life traces on AMD Athlon 64 3200+ (left) and on SGI Altix 3700 (right). Figures are normalized relative to STF.



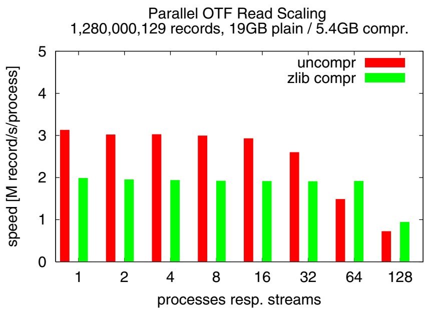


Fig. 4. Parallel input speed on SGI Altix 3700: total (left) and normalized (right)

On the Altix platform (Figure 3 right) all three examples behave similar to the small case on AMD 64 platform. Because of this machine's excellent I/O capabilities input speed is just not critical, not even for traces of 2.3 GB size. Because of that, uncompressed OTF is fastest for all three examples. It is followed by compressed OTF which is still faster than STF and VTF3. Nevertheless, OTF compression should be used by default because of the reduced storage footprint.

Parallel Read Performance. Finally, Figure 4 shows parallel performance for plain and compressed OTF on the Altix platform. It was evaluated with a very huge trace containing 128 processes and 1 280 000 000 records. This results in $\approx$ 19 GB uncompressed or $\approx$ 5.4 GB compressed total storage size. This same trace was stored with $n = 1 \ldots 128$ streams and read with n processes.

The left hand side of Figure 4 shows the aggregated total read performance (in million records per second), the right hand side the normalized counterpart (in million records per second per process).

The diagrams reveal a very nice scaling parallel input speed when reading uncompressed OTF. The sustained input speed of 3 000 000 records/s/process scales almost linearly up to 16 processes. At 32 processes the total input speed is saturated and the uncompressed read performance per process drops notably.

The normalized speed when reading compressed OTF reaches only $\approx 2/3$ of the uncompressed speed for small processor counts. When scaling to larger processor counts uncompressed OTF reading speed declines at > 16 processes. Compressed OTF reading keeps scaling linearly up to 64 processes! This relates to the average compression ratio $\approx 4 : 1$, i.e. $1/4$ input size.

For $n = 128$ processes even compressed OTF will overcharge total I/O bandwidth reducing the sustained input speed. Always, there is a point where total I/O performance limits input speed. As soon as this happens compressed OTF provides a big performance advantage.

5 Conclusion

OTF is a fast and efficient trace format library with special support for parallel I/O. It provides a convenient interface similar to existing trace format's ones. In addition it offers extended selective access along with some support features.

It achieves very good performance on single processor workstations as well as on massive parallel super computers. It delivers fast sequential and parallel access which scales very well depending on raw I/O capacity.

Experimental transparent blockwise ZLib compression allows to reduce storage size notably and improves parallel scalability on the same time.

OTF is available at [7] under BSD open source license that allows free usage for academic and commercial applications. Future work will include porting and tuning for additional platforms. Furthermore, a carefully extention of record types supported by OTF will be done according to user feedback. This might involve new MPI-2 related record types, for example.

Acknowledgment. Design and development of OTF has been done in cooperation with University of Oregon and Lawrence Livermore National Laboratory (LLNL) of University of California. Especially we would like to thank Prof. Allen D. Malony and Dr. Sameer S. Shende.

Furthermore, we are very grateful to the High Performance Computer group of ZIH, TU Dresden for their support with early I/O performance measurements on SGI Altix 3700. Special thanks to Michael Kluge and Thomas Blümel.

References

1. Ruth A. Aydt. The Pablo Self-Defining Data Format. Technical report, Department of Computer Science, University of Illinois, Urbana/Illinois, USA, 1994. `http://wotug.kent.ac.uk/parallel/performance/tools/pablo/`.
2. Holger Brunst, Allen D. Malony, Sameer S. Shende, and Robert Bell. Online Remote Trace Analysis of Parallel Applications on High-Performance Clusters. In *Proceedings of ISHPC'03 Conference*, Springer LNCS 2858, pages 440–449, 2003.
3. J. Chassin de Kergommeaux, B. de Oliveira Stein, and G. Mouni. Paje Input Data Format. Technical report, 2003.
4. Intel GmbH, Brühl, Germany. *Intel Trace Collector User's Guide*.
5. Guohua Jin and John Mellor-Crummey. Experiences Tuning SMG98 - a Semicoarsening Multigrid Benchmark based on the Hypre Library. In *Proceedings of the International Conference on Supercomputing*, New York, June 2002.
6. Jean loup Gailly and Mark Adler. *zlib 1.1.4 Manual*, March 2002. `http://www.zlib.net/manual.html`.
7. Hartmut Mix. Open Trace Format. `http://tu-dresden.de/zih_otf/`, 2006.
8. Stephan Seidl. VTF3 - A Fast Vampir Trace File Low-Level Library. personal communications, May 2002.
9. The ASCI Project. The IRS Benchmark Code: Implicit Radiation Solver. `http://www.llnl.gov/asci/purple/benchmarks/limited/irs/`, 2003.
10. University of Oregon. *TAU User's Guide*. `http://www.cs.uoregon.edu/research/tau/docs/newguide/`.
11. Felix Wolf and Bernd Mohr. EPILOG Binary Trace-Data Format. Technical report, Forschungszentrum Jülich, University of Tennessee, 2004. FZJ-ZAM-IB-2004-06.