Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 2115–2124

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

A generic attribute extension to OTF and its use for MPI replay

Andreas Knüpfer[a,*], Markus Geimer[b], Johannes Spazier[a], Joseph Schuchart[a], Michael Wagner[a], Dominic Eschweiler[b], Matthias S. Müller[a]

[a]*ZIH, TU Dresden, 01062 Dresden, Germany*
[b]*JSC, Forschungszentrum Juelich GmbH, 52425 Juelich, Germany*

Abstract

Data formats are important components for parallel event tracing tools used for debugging and performance analysis. This paper presents a generic extension mechanism for the Open Trace Format (OTF) that allows to add arbitrary data to the existing event data types. This avoids the need to frequently adapt the data format and the library API to accommodate new usage scenarios. The replay of MPI communication is demonstrated as a real-world example that requires an extension. Several other application scenarios that would benefit from additional data about certain event types are mentioned.

Keywords: Event Tracing, Open Trace Format, MPI Replay

1. Introduction

Event tracing is an established method for the analysis of application execution behavior. In particular, it is very valuable for parallel debugging and performance analysis. There is a range of established tools, from experimental research tools to stable product-quality tools, with their own event trace file formats, even though most are very similar. A certain degree of interoperability is provided by tools that read foreign formats. The Open Trace Format (OTF) tries to leverage this idea and is already supported by a number of tools. Yet, there is a dilemma whether to strictly focus the design of the format towards its main application or to make it flexible and generic. The cost will be either interoperability or efficiency. This paper proposes a generic extension as a solution, which allows almost any kind of additional data for existing or new event types without impairing efficiency for the predefined events.

The rest of this paper is structured as follows: The next section introduces event tracing in general, data file formats, common event types, and their properties. In particular, it addresses OTF and its extension strategy. As the central point, the generic extension mechanism is presented in Section 3, together with an efficient encoding method and the API additions. After that, the MPI replay of non-blocking point-to-point communication is shown as a detailed example how the extension can be used to accomplish new features. Further application scenarios are briefly mentioned. Finally, the paper is concluded with a summary and an outlook.

[*]Email address: `andreas.knuepfer@tu-dresden.de`

2. Event Tracing and the Open Trace Format

Event tracing is a method for recording detailed information about the run-time behavior of sequential or parallel applications. It monitors individual *events* (points of interest) and stores them as *event records* together with relevant properties of the event, for example fine-grained timing information. The result is called the *event trace* of the application and allows very detailed post-mortem analyses.

Event tracing can be employed for debugging [1, 2] or for performance analysis [3, 4, 5, 6]. Usually, it involves three steps: First, the *instrumentation* which is slightly altering the application in order to point out events during run-time. Second, the *run-time data collection* which acts upon the events, collects additional properties, and stores the entire event trace. Finally, the *analysis* that reads and processes it.

2.1. The Open Trace Format

The role of the trace data format in this process is to transport the data from the data collection to the analysis step. The Open Trace Format (OTF) [7] was developed by ZIH, TU Dresden in collaboration with the University of Oregon and the Lawrence Livermore National Lab (LLNL). It is provided to the community as open-source software under the BSD license.

OTF is a part of the Vampir tool suite for parallel performance analysis [3, 6, 8] and has been adopted by a number of other projects, for example Open MPI [9], Open|SpeedShop [10], g-Eclipse [11], and TAU [12]. All of these tools, like OTF itself, are designed with High Performance Computing (HPC) in mind. Therefore, they are oriented towards parallel operation and scalability. Besides the actual data format definition, OTF provides a read and write library with a convenient and powerful API as well as a number of support tools [13]. The reader and writer API is available for C, C++, and Python. OTF supports transparent compression and allows to distribute parallel traces over multiple files (called streams) which can be accessed individually or as one logical trace. Tools are available, e.g., for conversion to/from other formats, for merging/separating streams of a trace, for generating profiles, and for filtering [7].

2.2. Related Work

There are a number of other trace formats for parallel performance analysis. Some are very similar to OTF with respect to the design and the API, for example, the older Vampir Trace format (VTF) , the Structured Trace Format (STF) [14], the EPILOG format [15], or the TAU trace format [12]. None of them includes a generic feature like the proposed extension.

The Paraver format [16] uses a simplistic and static format which contains only tuples of numbers. Yet, the contents can be interpreted in a very generic way because the format defines very little semantics. However, this format is very different from all other formats and therefore only supported by the Paraver tools. The Pablo Self-Defining Data Format (SDDF) [17] has a different design approach to provide maximum flexibility by including the specification of the actual data records in the format itself. See also [18] for an overview of trace file formats.

2.3. Event Types and Properties

The Open Trace Format has a fixed set of event types with appropriate properties, where appropriate means for the typical use case it was designed for: performance analysis of sequential and parallel applications including visualization, (semi-)automatic detection of performance flaws, and statistical evaluation. Therefore, it contains event types to record sequential and parallel aspects, including enter and leave of subroutine calls, performance counter samples, I/O operations, message passing (MPI), and multi-threading (OpenMP, PThreads). It provides fine-grained timing information for all event types and the assignment of events to a process/rank/thread/etc. The most prominent event types are shown in Table 1.

2.4. Extension Strategies

The event set of a trace format is not supposed to be very dynamic. Nevertheless, new developments did require adaptations, and will continue to do so. One example from the hardware side was the Cell BE architecture with a new communication scheme [19]. Another example from the software side was (or will be) the introduction of non-blocking collective operations to the MPI standard [20].

Table 1: OTF's most prominent event types with their predefined properties.

Enter, Leave	time, process, function
Send, Receive	time, sender, receiver, communicator, tag, bytes transferred
Counter	time, process, counter, value
BeginCollectiveOperation	time, process, collective operation, matching ID, communicator, root process, bytes sent, bytes received
EndCollectiveOperation	time, process, matching ID
BeginFileOperation	time, process, handle
EndFileOperation	time, process, file, handle, operation, bytes transferred

An event trace data format needs to incorporate new developments, otherwise it would become useless and soon be replaced. Yet, it needs to be stable with respect to the functionality and the API. Frequent changes would not encourage developers to build stable tools on top of it. While additions might be feasible, reductions are very difficult because they might break third party applications. Therefore, the extension strategy of OTF has been very conservative. New features are added hesitantly and only after thorough consideration. This is to maintain a clear and consistent interface that is not constantly growing. However, it is hindering all but the precisely intended purposes, because new features are introduced slowly. Experimental implementations of potential new features cannot be tested with the official release, nor can features for very special purposes that will never be part of the official version. Still, the alternative of a frequently changing format would inflict consequences on the tools using the format.

3. Generic Extensions for the Open Trace Format

The generic extension presented in this section is intended to solve the abovementioned dilemma. With this extension, nearly any kind of data can be transported besides the existing event set without the need for further API changes. It is usable for stable applications as well as for experimental purposes. Still, the generic way of storing auxiliary information is not intended to replace the existing model for the established event types.

3.1. Key-Value Lists as Generic Properties

The generic solution to carry additional properties for trace events are lists of key-value pairs. For every event there will be such a list in addition to the existing properties. The list object can contain any number of key-value pairs (or none) such that an arbitrary number of extended properties can be added to an event. The key part of the pairs is used to identify the extended properties. It consists of a 32 bit unsigned integer that refers to a key definition.

The key definition record provides a name that can be recognized by consuming tools. It is suggested to use a namespace convention where names are prefixed in C++ style with the tool's name and double colons, for example "`vampir::`". The application/tool writing an OTF file must guarantee that there are no conflicting key definitions. This is no limitation as keys should be commutable. However, both conditions are not checked or enforced by the OTF library.

The value part may contain signed or unsigned integers (8, 16, 32, or 64 bit), floating-point values (32 or 64 bit), or byte arrays of limited length `OTF_KEYVALUE_MAX_LENGTH`[1]. This covers the same value domain as all existing predefined properties of OTF events including strings. Yet, it is not indented to carry huge amounts of data such as the contents of arrays.

[1]The length limit for byte arrays shall guarantee an efficient internal data handling inside the OTF library. The total data amount is (practically) unlimited, as successive keys can be used to transport the data in chunks.

3.2. The Key Definition Record Type

A further addition is a key-value definition record which allows to identify a key from a name and a descriptive text. The intended data types for the associated values can be specified corresponding to basic C data types. This is not checked by the OTF library at this stage, however. The definition record is designed like regular definition records with its own write and call-back routines according to the following record type specification:

```
DefKey( uint32_t key, char* name, char* description, OTF_Type type )
```

3.3. Encoding of Key-Value Lists in OTF

OTF stores all events as *event records* in multiple files (called streams). The records within a stream are chronologically sorted. A single stream contains one or more processes/threads but every process/thread belongs exclusively to one stream. Within the streams, the timestamp and process specifications, which are required for all event records, are not stored as part of the event records but as separate records which are handled internally: the timestamp record specifies the time for all following events in the stream until the next timestamp, redundant timestamps for successive events are suppressed (accordingly for process records). For more details about the internals and the encoding of OTF see [7, 13].

On the encoding level, the key-value pairs are stored as separate key-value records. The key-value records are associated to events via their position by placing them directly before the regular record. In contrast to timestamp and process specifications, a key-value record is only connected to the next record, but not to all following ones.

```
TIMESTAMP 1000123a00
PROCESS 2a
ENTER 5
TIMESTAMP 1000123f00
PROCESS 2a
LEAVE 5
...
TIMESTAMP 1000124000
PROCESS 2a
KEY 3 TYPE 2 VALUE deadbeaf
KEY 4 TYPE 1 VALUE ff
ENTER 6
TIMESTAMP 1000124100
PROCESS 2a
KEY 5 TYPE 1 VALUE 0
LEAVE 6
```

Figure 1: Schematic encoding of subroutine calls in OTF: Enter and Leave records without (top) or with additional key-value records (bottom).

During writing, the key-value records can simply be written before the associated event record. During reading, key-value records are collected by the current stream. As soon as a regular event record is encountered, it is delivered to the consumer via a call-back routine together with the collected key-value pairs. Right after the return of the call-back routine, the list object is cleared and begins collecting the next key-value pairs. This makes the extension backward and forward compatible. It allows to implement the writing and parsing of key-value records separately from that for regular event records.

Figure 1 shows an example of the encoding principle (note that all numbers are given in hexadecimal notation). At first, there is a normal call to a subroutine with the identifier `'5'` denoted by an Enter and a Leave event on process `'2a'` with given timestamps. Next, there is a call to subroutine `'6'` with process and timestamp specifications. In addition, the Enter event carries two key-value records: One with key `'3'` and a byte array containing `'deadbeaf'` as the value and one with key `'4'` and an integer value `'ff'`. These could be the values of the subroutine's first and second arguments. The Leave event carries a single key-value record with key `'5'` and the integer value `'0'`. This could be the call's return value.

3.4. The No-Op Event Type

So far, arbitrary extensions can be made to existing event types. For further flexibility, a placeholder event type is introduced called `NoOp`. It consists only of a timestamp and a process with the only purpose to carry key-value pairs when there is no other event to attach them to.

3.5. Extensions to the OTF Record API

The OTF writer interface introduces an extension to the event writer routines for all event types. It adds a key-value list (`kvlist`) as last argument, which is allowed to be `NULL` or an empty list. The original routine is not replaced, though, because this would break existing code:

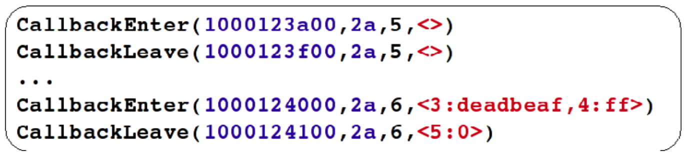
```
OTF_Writer_write<NAME>( writer, time, process, ...  )
OTF_Writer_write<NAME>_KV( writer, time, process, ...  , kvlist )
```

The reader interface of OTF delivers events by invoking call-back routines that the consumer needs to register beforehand. The call-back routines need to accept the correct arguments which is not checked by the C language. Corresponding to the write calls, the call-back routines are extended by a read-only key-value list object like the following:

```
Callback_handle<NAME>( user_data, time, process, ...  , kvlist )
```

The call-back mechanism can not distinguish between user routines with or without the final key-value list argument, however. For a correct call-back routine that is missing only the last argument, it is silently ignored. Therefore, this is backward compatible[2].

All in all, the user interface for extended event records is very similar to the familiar record interface even though the low-level handling is different. The user perspective of the example in Figure 1 as provided by the OTF reader API is shown in Figure 2: OTF will deliver four events via call-back routines. The timestamps, the process specifications and the function identifier are given for all Enter and Leave events. The last two contain additional key-value pairs, while the first two have an empty list as the last argument.



```
CallbackEnter(1000123a00,2a,5,<>)
CallbackLeave(1000123f00,2a,5,<>)
...
CallbackEnter(1000124000,2a,6,<3:deadbeaf,4:ff>)
CallbackLeave(1000124100,2a,6,<5:0>)
```

Figure 2: OTF reader call-backs for the example in Figure 1.

3.6. The OTF Key-Value-List API

As the final piece, there is a new data type and its associated methods in the OTF API that fall into three categories (see also Table 2). First, there are routines for creating and destroying objects. Second, there are modification methods which will typically be used during writing. And third, there are read-only methods for querying the contents.

4. Example Application Case: MPI Replay

In this section, an MPI replay implementation is presented on top of OTF and the generic trace format extension with key-value pairs. This is an interesting use case because it requires a number of additional properties. Thus, it demonstrates the capability to support a broad range of information of different kinds.

In particular, the replay of MPI activities is useful for a number of purposes, either with or without the original application. First of all, it can be used as an MPI benchmark closely mimicking the original MPI application, including all MPI calls and the "waiting time" in between. This would allow to test the MPI performance of different machines without the need for the original application. It might be useful because of software porting issues or because of restrictions with proprietary or confidential software or data. Furthermore, it would allow to use modified or synthetic traces. This could be most useful to test the potential MPI behavior of an application before actually porting it. In [21] this concept is presented more elaborately.

[2]This should be true for (almost) all platforms and compilers and should be good enough for the transition period.

Table 2: The central methods for the Key-Value-List data type. The `<Type>` placeholder stands for 8/16/32/64 bit signed/unsigned integers, single/double precision floating point types, or bytes arrays of limited length.

```
OTF_KeyValueList* OTF_KeyValueList_new();
uint8_t OTF_KeyValueList_close( kvlist );
bool appendPair<Type>( kvlist, key, <type> value, [uint32_t len] );
bool appendList( kvlist, source_list );
void reset( kvlist );
bool hasKey( kvlist, key );
type getType( kvlist, key );
bool getValue<Type>( kvlist, key, <type>* value, [uint32_t* len] );
bool getKeyByIndex( kvlist, index, key_ptr );
```

Another application scenario for MPI replay is the debugging of race conditions. It will record all MPI operations that may contribute to non-deterministic behavior of the application. Later, it will be used for a replay together with the original application. Instead of plainly re-executing the application, a special MPI layer will force the pre-recorded outcome of all race conditions. Like before, a modified version of the original trace could be used as well. This allows to investigate and debug the non-deterministic behavior in a deterministic manner [2, 22].

A very interesting way of MPI replay is used by the Scalasca toolset for automatic performance analysis [4]. It performs a replay of all communication calls with the original number of MPI ranks. The replay messages are used to transmit local performance data to communication partners which allows a distributed automatic analysis of timings.

As a final example, a certain kind of MPI replay is required for parallel fault tolerance [23]. When uncoordinated parallel checkpoints are used, a partial replay of recorded MPI communication is required for the failing ranks. However, this differs very much from the application scenarios mentioned above.

4.1. MPI Replay Activities

The basic idea of replaying MPI operations from an event trace is rather trivial. The replay engine is reading the event trace in parallel, associating the current MPI ranks to the ones in the event trace. As soon as a relevant event is encountered, the corresponding action is triggered. This mode is working fine with existing OTF traces for blocking communication, as every trace event contains all the necessary information. Table 3 gives an overview about the actions required for the most important event types.

Table 3: Replay actions for the most important event types for blocking MPI communication.

Enter/Leave	- for MPI calls: add prior waiting time to emulate the original timing - for other calls: optionally insert artificial enter/leave events
Send/Receive	- execute a corresponding blocking send/receive call - allocate and free the required buffers
Counter	- ignore
Begin/End Coll. Op.	- execute a corresponding collective operation - allocate and free the required buffers
Begin/End FileOperation	- execute corresponding (MPI) I/O call - manage file handles between open, access, and close

4.2. Additional Properties for Non-Blocking Communication

The current arrangement of OTF records is insufficient for the replay of non-blocking point-to-point communications as shown by the example in Figure 3. Note that there can be multiple `MPI_Test` calls as potential finalization

```
MPI_Isend( &sum, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, &request );
do_something_else();
MPI_Wait( &request, &status );
```

```
MPI_Irecv( &sum, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &request );
while ( 1 ) {
    flag= 0;
    MPI_Test( &request, &flag, &status );
    if ( flag ) break;
    do_something();
}
```

Figure 3: Code snippets for non-blocking MPI communication for the example in Figure 4: `MPI_Isend` plus `MPI_Wait` at the sender side (top) and an alternative of `MPI_Irecv` plus a series of `MPI_Test` calls at the receiver side (bottom).

points for a single non-blocking `MPI_Irecv` call. Figure 4 shows one possible manifestation of the examples in an event trace. The Send event (S) is placed between the Enter and Leave for `MPI_Isend`. Yet, the Receive event is not between Enter and Leave for `MPI_Irecv` but inside the `MPI_Test` call.

This arrangement is most useful for performance analysis showing the logical points in time where the message is sent or received and allows to give a conservative estimate of the message speed. For the sake of replay, the information from the `RecvMsg` event, which is not available before the final `MPI_Test` call, would be needed during the `MPI_Irecv` subroutine. This can be achieved by extending Enter, Send, and Receive events with key-value pairs like the following:

Send: If a Send event stands for a non-blocking send, flag it with a special key-value pair and link it to its finalization event with a request ID in a second key-value pair (Keys 1 and 5).

Receive: If a Receive event stands for a non-blocking receive operation, then mark it and give a request ID to the start event of this message (Keys 1 and 5).

Enter I: If an Enter event is for a message finalization call like `MPI_Wait`, `MPI_Test` or their variants, then give a special key-value pair as a flag and another to provide a request ID to link to the start operation of the current message (Keys 1 and 5).

Enter II: If an Enter event is for `MPI_Irecv`, then mark it (Key 1) and give all information required to replay the call (sender, communicator, tag, length with Keys 2, 3, 4, 6 in the example). Also, link to the associated finalization event via a request ID (Key 5).

The request IDs are needed to identify the matching start and end events of the messages. Note, that special considerations are necessary for the outcome of `MPI_Test` calls during the replay. Since it may differ from the original run, two special cases need to be handled: First, if the test succeeds earlier than expected, then all following tests with the same request ID have to be ignored. Second, if the final test (that with the actual receive event inside) is not successful, then the replay engine needs to wait for the message, because there will be no further test calls.

This example illustrates the extended event properties. For the complete functionality of MPI replay, further extended properties are needed, e.g., for operations with wildcard arguments or one-sided MPI calls. They can be implemented in a similar manner as shown above. Yet, all of the extended properties would be considered overhead for the MPI performance analysis process, which is the original purpose of OTF. Therefore, those properties would not be made standard properties for OTF event types. Creating a separate trace file format for replay traces would cause a lot of redundant work. However, replay traces are still interesting for performance analysis[3].

[3]One could facilitate the replay engine to explicitly write an event trace of the replay run. Yet, it turned out to be much easier to use the automatic instrumentation of VampirTrace [24] for the replay program, generating a trace like for any other target application.

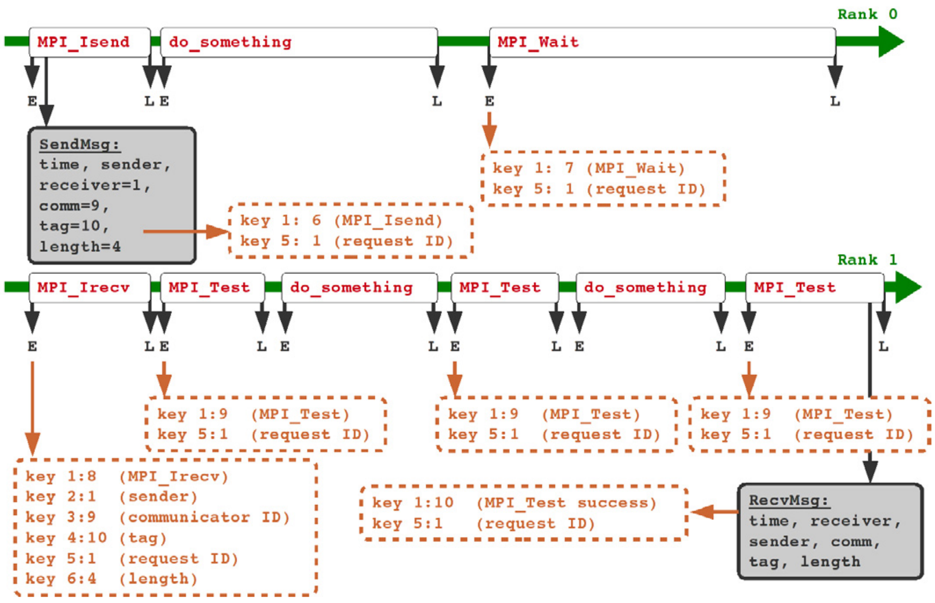


Figure 4: Example event sequence for a non-blocking MPI message exchange from the sender rank 0 (top) to the receiver rank 1 (bottom), see also Figure 3. It indicates the Enter (E), Leave (L), Send, and Receive events. Note that the RecvMsg properties are not available before the final `MPI_Test` call. The extended properties (dashed) annotate the properties at the point where they are needed for replay. Note that the keys are chosen arbitrarily and have no special meaning.

5. Further Use Cases for Extended Trace Properties

Besides the replay example described above, there are a number of other use cases where key-value pairs can be employed. Some of them were considered as official changes for OTF, but rejected because of a major change in the API with too little merit for the average OTF user.

Send-Receive Mapping: The mapping between associated Send and Receive events is important for visualization and for computing the transfer speed. Yet, it needs to be computed every time a trace is read. It could be pre-computed and stored persistently with key-value pairs.

Function Call Arguments: For certain function calls it would be interesting to know some of the call-time arguments which can be transported by key-value pairs. For example, it could be worthwhile knowing the dimension when calling routines for matrix or vector operations.

Additional Call Stack Information: The Sun Studio Performance Tools [25] use VampirTrace in MPI-only mode. Still they want the complete call stack for MPI calls in order to differentiate the call sites. Currently, this is written to special comment records. It could also be solved with key-value pairs, where keys for the call levels refer to the existing function definitions.

Clock Correction Information: During parallel event trace collection local clocks are used for tracking time. It turns out, parallel clocks are in general far from being adequately well synchronized in order to evaluate the parallel run-time behavior. A periodic timer synchronization method can generate offset and drift information that allows to transform all local clocks into a sufficiently synchronized global clock as a post-mortem operation [26]. The correction information is currently stored outside of OTF but could be embedded in the trace with key-value pairs.

Alternative Lamport Clock: The DeWiz and gEclipse tools support an alternative timing using integer Lamport clocks in addition to the wall clock timers [2]. The Lamport clock can be added with a key-value pair to every single event in the trace.

Certainly, there are many further ways of using extended information transported by key-value pairs. It may be for performance analysis or for entirely different purposes.

6. Evaluation

The additional data transported by key-value pairs will influence the storage volume, of course. In Figure 5 exemplary OTF data volumes are shown with 0, 1/8, 1/4, 1/2, 1, 2, 4, and 8 key-value pairs per regular event. It comes from a synthetic test that covers very sparse to very excessive usage of key-value pairs in four OTF streams [7] of 100 MB each (before adding key-value pairs) using the default Zlib compression. Next to this, the read and write durations are given. Obviously, they correspond closely to the file sizes. The test platform was an up-to-date desktop PC.

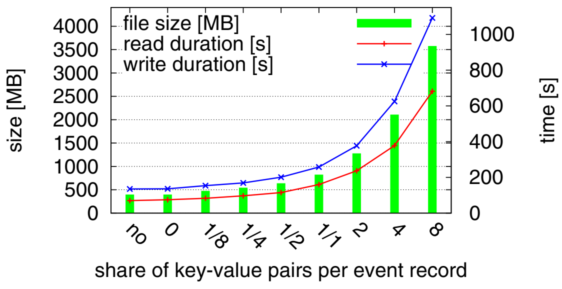


Figure 5: OTF data volume and read/write durations with different ratios of key-value pairs per event. The durations correlate very closely to the data volumes.

7. Conclusions and Future Work

This paper presented a generic extension which enables the Open Trace Format for many new application scenarios. The extension allows a lot of flexibility for storing additional data related to existing or new types of events. However, it will not convert the existing event types to generic ones for the sake of conceptual clarity and efficiency. As a non-trivial example, the implementation of a replay mechanism for non-blocking MPI communication has been presented.

Future work based on the presented results shall make the extension available to the OTF user community. The generic key-value pair extension is planned to be part of the next major release of OTF. The example implementation of the MPI replay mechanism will be enhanced to become part of the next release of VampirTrace. This includes an optional recording mode which records replay capable traces and a replay engine for MPI performance benchmarking. Also, we want to encourage and support the implementation of further use cases as mentioned in Section 5.

This will also reveal whether the extension should have been taken one step further by allowing key-value pairs for definition records, which is currently ruled out. This would in particular allow key-value pairs for the definition of (other) keys, which might be either useful or unnecessary.

References

[1] T. Hilbrich, M. S. Müller, B. Krammer, MPI Correctness Checking for OpenMP/MPI Applications, International Journal of Parallel Programming 37 (3) (2009) 277–291. doi:10.1007/s10766-009-0099-4.

[2] D. Kranzlmüller, M. Scarpa, J. Volkert, DeWiz - A Modular Tool Architecture for Parallel Program Analysis, in: Parallel Processing, Proc. 9th Intl. Euro-Par Conference, Springer LNCS 2790, Klagenfurt, 2003, pp. 74–80.

[3] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, W. E. Nagel, The Vampir Performance Analysis Tool-Set, in: M. Resch, R. Keller, V. Himmler, B. Krammer, A. Schulz (Eds.), "Tools for High Performance Computing", Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, Springer-Verlag, Stuttgart, Germany, 2008.

[4] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker, B. Mohr, The SCALASCA Performance Toolset Architecture, in: Proceedings of the International Workshop on Scalable Tools for High-End Computing, 2008.

[5] J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, DiP: A Parallel Program Development Environment, in: Proc. of 2nd International EuroPar Conference (EuroPar 96), Lyon/France, 1996.

[6] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, W. E. Nagel, Developing Scalable Applications with Vampir, VampirServer and VampirTrace, in: C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.), Parallel Computing: Architectures, Algorithms and Applications, Vol. 15 of Advances in Parallel Computing, IOS Press, 2007, pp. 637–644.

[7] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. E. Nagel, Introducing the Open Trace Format (OTF), in: V. N. Alxandrov, G. D. Albada, P. M. A. Slot, J. J. Dongarra (Eds.), 6th International Conference on Computational Science (ICCS), Proceedings Part II, Vol. LNCS 3992, Springer, Reading, UK, 2006, pp. 526–533.

[8] M. Jurenz, The VampirTrace documentation, `http://www.tu-dresden.de/zih/vampirtrace/` (2009).

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings of the 11th European PVM/MPI Users Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[10] M. Schulz, J. Galarowicz, W. Hachfeld, Open|SpeedShop: Open Source Performance Analysis for Linux Clusters, in: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2006, p. 14.

[11] C. Klausecker, T. Köckerbauer, R. Preissl, D. Kranzlmüller, Debugging MPI Programs on the Grid using g-Eclipse, in: 2nd Parallel Tools Workshop, Stuttgart, Germany, 2008.

[12] S. Shende, A. D. Malony, The TAU Parallel Performance System, International Journal of High Performance Computing Applications (ACTS Collection) Vol. 20 (2) (2006) 287–311.

[13] A. Knüpfer, The OTF library documentation, `http://www.tu-dresden.de/zih/otf` (2009).

[14] Intel GmbH, Brühl, Germany, Intel Trace Collector User's Guide, document number 318119, `http://www.intel.com/` (2007).

[15] F. Wolf, B. Mohr, EPILOG Binary Trace-Data Format, Tech. Rep. FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich (May 2004).

[16] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain, PARAVER Version 3.0 Tracefile Description (June 2001).

[17] R. Aydt, The Pablo Self-Defining Data Format, Tech. rep., University of Illinois at Urbana-Champaign (Feb 1993).

[18] A. Knüpfer, Advanced Memory Data Structures for Scalable Event Trace Analysis, Ph.D. thesis, TU Dresden, Suedwestdeutscher Verlag fuer Hochschulschriften, ISBN 978-3838109435 (Sept 2009).

[19] D. Hackenberg, H. Brunst, W. E. Nagel, Event Tracing and Visualization for Cell Broadband Engine Systems, in: E. Luque, T. Margalef, D. Benítez (Eds.), Euro-Par 2008, LNCS 5168, Springer-Verlag, Las Palmas, Gran Canaria, Spain, 2008, pp. 172–181.

[20] T. Höfler, P. Gottschling, A. Lumsdaine, Leveraging Non-blocking Collective Communication in High-Performance Applications, in: Proceedings of the 20'th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08).

[21] D. Böhme, M.-A. Hermanns, M. Geimer, F. Wolf, Performance Simulation of Non-Blocking Communication in Message-Passing Applications, in: EuroPar 2009 Workshop Proceedings, 2009, (to appear).

[22] A. Bouteiller, G. Bosilca, J. Dongarra, Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Vol. Springer LNCS 4757, 2007, pp. 297–306.

[23] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, F. Cappello, Improved message logging versus improved coordinated checkpointing for fault tolerant MPI, in: CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing, IEEE Computer Society, Washington, DC, USA, 2004, pp. 115–124.

[24] M. Jurenz, R. Brendel, A. Knüpfer, M. S. Müller, W. E. Nagel, Memory Allocation Tracing with VampirTrace, in: International Conference on Computational Science (2), 2007, pp. 839–846.

[25] M. Itzkowitz, The Sun Studio Performance Tools, Tech. rep., Sun Microsystems Inc (2005).
URL `http://developers.sun.com/solaris/articles/perftools.html`

[26] J. Doleschal, A. Knüpfer, M. S. Müller, W. E. Nagel, Internal Timer Synchronization for Parallel Event Tracing, in: A. Lastovetsky, T. Kechadi, J. Dongarra (Eds.), PVM/MPI, Vol. 5205, Springer-Verlag, 2008, pp. 202–209.