Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries

Conference Paper · January 2012

DOI: 10.3233/978-1-61499-041-3-481

Michael Wagner
German Aerospace Center (DLR)
29 PUBLICATIONS **308** CITATIONS
SEE PROFILE

Markus Geimer
Forschungszentrum Jülich
50 PUBLICATIONS **1,120** CITATIONS
SEE PROFILE

Wolfgang E. Nagel
Technische Universität Dresden
258 PUBLICATIONS **3,368** CITATIONS
SEE PROFILE

Open Trace Format 2 The Next Generation of Scalable Trace Formats and Support Libraries

Dominic ESCHWEILER [a], Michael WAGNER [b], Markus GEIMER [a], Andreas KNÜPFER [b], Wolfgang E. NAGEL [b], Felix WOLF [c]

[a] *Jülich Supercomputing Centre, 52428 Jülich, Germany*

[b] *Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany*

[c] *German Research School for Simulation Sciences, Laboratory for Parallel Programming, 52062 Aachen, Germany*

Abstract. A well designed event trace data format is the basis of all trace-based analysis methods. In this paper, we introduce the Open Trace Format Version 2 (OTF2). It is a major re-design based on the experiences of its predecessor formats, the Open Trace Format (Version 1) and the EPILOG trace format. It comes with a new file encoding, a close integration in the trace recording system Score-P, and a number of improvements for performance and scalability. Besides the actual format, it consists of a read/write support library with a powerful API plus supportive tools, which are distributed as Open Source software. Furthermore, OTF2 will serve as a joint data source for the analysis tools Scalasca and Vampir in the near future.

Keywords. Tools Tracing Scalability

Introduction

Applications which are supposed to effectively utilize the enormous computational resources of today's HPC systems must meet very high requirements. Developing such applications demands knowledge of the complex systems and underlying hardware, of parallel programming paradigms, and the behavior of the own source code. These tasks become more and more complex and can hardly be performed without support of appropriate tools.

Two common approaches to analyze the performance of applications are *profiling* and *event tracing*. Profiling is the gathering of summarized information about different performance metrics during runtime. While offering a good starting point to understand performance problems, it does not provide further insight into the application's dynamic behavior. In contrast event tracing tools record all events that are of interest for later examination, together with the time they occured and a number of event type specific properties during application runtime. Typical events are entering and leaving of functions or sending and receiving of messages.

Thus, trace-based analysis tools have access to the detailed dynamic application behavior and are therefore able to offer a much higher level of insight into occurring performance problems than profiling tools. However, the amount of collected information can be tremendous and usually results in several hundreds or thousands of megabytes of data per process. Therefore tools need highly memory efficient event trace formats and highly scalable access libraries to manage all the data. With the Open Trace Format 2, we present an event trace format and associated libraries that addresses those high demands.

In Section 1, we give an overview over existing event trace formats and their general design. After that, we discuss the necessity of a new event trace format and the goals we are trying to achieve with its novel design (Section 2). In Section 3 we describe implementation concepts and details. We present some first test results showing the performance of the OTF2 library in comparison with other well-established event trace reader and writer libraries in Section 4) Finally, we draw first conclusions in Section 5 and give an outlook on future work in Section 6.

1. Related Work

Trace data formats always have been an important component for event trace based tools, but they usually received very little attention. There are a number of trace file formats in the HPC performance analysis realm, which are mostly part of specific tool infrastructures, but rarely distributed on their own.

Most formats are very similar in design, but differ in event type details, their APIs, the encoding details, etc. Typically, all come with a reader and writer library and a defined API. Thus, the formats are essentially the APIs and libraries, while the actual file encoding format is hidden from the user. All formats are constructed from basic *event types*, e.g., for entering or leaving subroutine calls, sending or receiving messages, I/O operations, and many more. The content of the trace files are individual *event records* for all events of further interest that happened during runtime. They contain a time stamp, a process/thread identifier, and further type-specific properties. Usually, the events have to be sorted in chronological order, either per process/thread in multiple files or globally in a single file. In addition to *event records*, there are also *definition records* which provide general information like timer resolution, process names, and mapping tables.

Examples for trace file formats are the SLOG2 [1] format, the Dewiz trace format [2] and the TAU trace format [3], which are related to the tools of the same name. The format of Paraver, which also can be used with the Dimemas tool, is a very generic event trace format with basic and simple record types [4]. The Structured Trace Format (STF) from Intel Corp. is proprietary and binary and serves as the format for the Intel Trace collector tool [5].

The Open Trace Format (OTF) and the EPILOG trace format are the predecessors of the new Open Trace Format Version 2 (OTF2). OTF is the default format of the Vampir tool set [6] and EPILOG is used by Scalasca [7]. OTF was developed by Technische Universität Dresden in cooperation with the TAU tools group from University of Oregon and the Lawrence Livermore National Labs. It is widely adopted, for example by the Microsoft HPC Server event tracing infrastructure and the former Sun Studio Performance Tools.

Besides the traditional file formats, there are novel approaches for compressed storage of event trace data. The ScalaTrace tools use a form of regular expressions to represent repeated trace sequences in a compressed way [8]. It can achieve substantial data compression but allows only very coarse statistical timing information. The Complete Call Graph (CCG) approach uses a generalized call tree, which contains all events including time stamps [9].

2. Motivation and Design Goals

The trace format and the related reader and writer library are the basis of every trace-based analysis method. One of the major flaws of the predecessor formats of OTF2 (OTF and EPILOG) was the missing compatibility between the event trace formats of Vampir and Scalasca. Although there is already a close cooperation in development and training activities on the part of the analysis tools (e.g. [10]), the measurement tools and formats are separate up to now. On the one hand, Vampir is nowadays able to read EPILOG traces, but not with the full set of features. On the other hand, Scalasca is not able to read OTF traces, because it has much higher constraints for the semantics of the trace data. This led to situations where a user of both tools had to record the trace data twice to be able to use Vampir and Scalasca to analyze the same application. Given that a supercomputer nowadays can have several thousands of processors and often users of such tools want to analyze an application at higher scales, recording the trace twice means a tremendous waste of computational resources. Thus, the most important requirement for the new OTF2 library and format was, that it can fulfill the requirements of both tools. So users can use one and the same measurement infrastructure for both.

The second requirement after interoperability was scalability. Today, where a supercomputer can operate several thousands of CPUs, developer tools need to be scalable. To achieve this, every component of such tools needs to be scalable, too. For the trace reader and writer components there are two very important dimensions of scalability. First, the number of supported execution locations[1] like processes and threads, and second the scalability in time, which means the number of records per location that can be stored without severely modifying the execution behavior of the measured application. Events for each location are stored in a separate data stream, which needs to be stored in a file afterwards. But, today's file systems are not able to handle as many files handles as there are processes or threads. Therefore, we plan to use SIONlib [11] for achieving parallel scalability, especially with respect to file handles, and a couple of compression steps for scalability in time. Compression steps in this context means that the OTF2 library is able to already compress data during runtime and later on further compress the data that is stored to a file system.

The third requirement was to integrate much more functionality right into the trace library to provide better performance and usability than the former formats. For instance, OTF does not provide an API to write event records into a memory buffer. Instead it can handle a given buffer of events and has, therefore, an API to write it to disk in the right OTF formatting. On the other hand, EPILOG does provide an API for writing events into a memory buffer, but the buffer itself has to be allocated and written to disk by the user. In contrast, OTF2 provides an external API for event record writing as well as internal layers for memory management and file writing. Thereby the internal layers are hidden from the user as much as possible, to make the API transparent to internal

[1] In the further context a *location* always means an execution location like a process or a thread.

changes and easy to use. Hiding the buffer management from the programmer also gives us the possibility to introduce an online compression, which results in less buffer flushes than before. Reducing the number of buffer flushes is a major improvement, because they usually lead to a huge bias for further measurement, typically rendering the resulting trace useless for further analysis. Additionally, a new approach should be capable to read a trace backwards, to avoid that a tool, which uses the new library, needs to take care of this.

The last main requirement for OTF2 was to make the library easily extendable for new features. Therefore, OTF2 is designed in an modular fashion with clearly separated components that can be exchanged. For example, it is easily possible to add a number of alternative strategies for file writing like SIONlib, hierarchical directory structures, or network sockets. In addition, it is also possible to rapidly extend the format by generating most parts of the external API. This gives us the opportunity to introduce new record types into the format with a minimum of effort, just by their definition and a re-generation of the interface afterwards.

3. Implementation

This section describes some details of the OTF2 implementation. We extended the concepts of previous solutions with several features to enhance scalability and usability. OTF2 consists of two parts – a binary file format specification and a library which can read and write OTF2 traces.

3.1. File format

An OTF2 trace always consists of a set files, which is called an OTF2 archive. The different file types and their purpose can be described as follows (numbering is according to Figure 1):

1. *Anchor file* : The anchor file contains meta data, which is related to the archive organization.
2. *Global definition file* : The global definition file stores all definitions that are equal for all locations. Thereby, a location is a place where events of one execution instance are recorded (e.g. a thread).
3. *Local definition file* : A local definition file holds all definitions that are only related to a specific location. Additionally, these files store mapping information to translate local identifiers to their global counterparts.
4. *Local trace file* : A local trace file stores all events that were recorded on the related location.

The sizes of trace archives generated on todays supercomputers are too huge to be copied, moved, or rewritten in a reasonable amount of time. Therefore, OTF2 maps all local identifiers to global ones on-the-fly during reading. Furthermore, the data is stored in the same endianess like the machine where the trace was recorded. For portability, endianess conversion is performed automatically while reading if necessary.

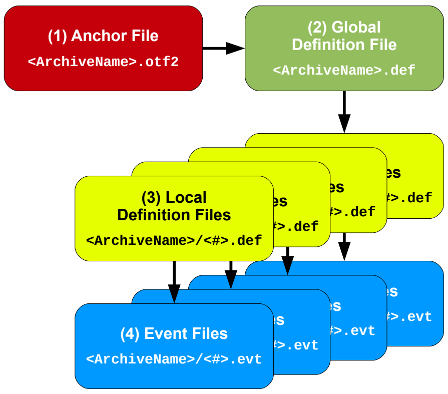

Figure 1. File system layout of an OTF2 archive.

3.2. *Infrastructure of the OTF2 library*

Fig. 2 shows the OTF2 library consisting of four layers, which are responsible for archive management (layer 1), record representation (layer 2, see also Section 3.2.1), memory representation (layer 3, see also Section 3.2.2), and file representation (layer 4, see also Section 3.2.3). It also integrates a plug-in infrastructure to integrate reader components for other formats than OTF2.

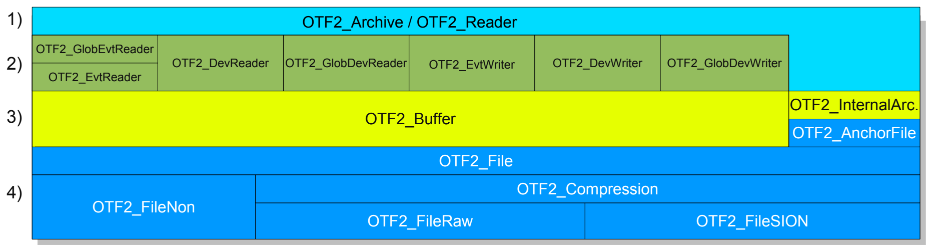

Figure 2. Layout of the OTF2 library.

To use OTF2, first all required reader or writer objects need to be requested from the central archive management. The trace data itself can then be stored and accessed by using these components, representing the record layer. To perform memory operations, they use the memory layer which consists of two components: the internal archive component, which is a memory representation of a trace's meta data, and the memory buffer component, which does the encoding and file system interaction. The latter uses the lowest layer for requests to the file system as well as for compression, consisting of several different substrates which are fully transparent to the memory layer.

3.2.1. Record representation

Nearly every component of the record layer serializes records into a sequence of atomic reads or writes of single variables. These reads or writes are performed with the buffer component (see Section 3.2.2) of the memory layer. In reading mode, every record read triggers a callback function to provide all record information to the user. Mapping tables from local definition files are transparently applied, so that only global identifiers are visible to the reader.

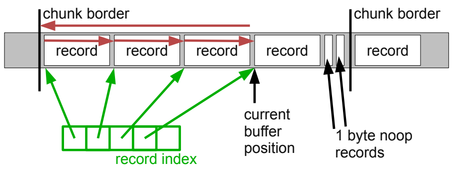

Figure 3.: The currently used chunk is indexed if backward reading is requested.

An automatic indexing is needed for backward reading and seeking (see Figure 3), because the variable size of attributes, caused by the internal runtime compression (Section 3.2.2), circumvents a direct calculation of start addresses of a targeted record. The records are organized in smaller units, called chunk, to avoid indexing the entire trace which would be very costly. These chunks provide fixed starting points for reading and are distributed over the whole trace. This reduces runtime and memory overhead if backward reading is just used for reading a few records backwards, compared to a solution which does indexing for the complete trace at once.

3.2.2. Memory representation

The most important part of the memory representation layer is the buffer component. It implements functions for atomic reads and writes, a leading zero compression, the chunk controlling, and it handles automatic flushing into a file if the available memory is exceeded.

The OTF2 library uses two techniques to avoid storing unnecessary data and to achieve a compact representation already at runtime. First, it stores the time stamp only once for sequences of events with an identical time stamp. Second, OTF2 removes all higher value zero bytes from integer attributes and stores the variable width into the first byte followed by this compressed sequence. The buffer component is also capable of using an external memory management. For example the measurement environment Score-P [12] uses OTF2 with a memory pool dynamically distributed over all threads of the same process. This enables a better resource distribution on imbalanced threads than static memory assignment.

The buffer component is also used to read a trace. The reading mode is designed to reduce memory usage and to enable future support for data prefetching. The minimization of the memory consumption on reading is needed for use-cases where the trace consumer runs on a much smaller computer than the trace producer. The OTF2 buffer component achieves this by holding only three chunks in memory at any time: the current, the previous and the next.

3.2.3. File representation

The file representation layer implements functions to abstract from operating system dependent file handling routines and different encoding libraries. This layer is also shown in Figure 2 as layer four. It currently supports POSIX file interaction and no file system interaction for later in-memory-analysis. In the future, we plan to integrate the SION library [11] to achieve high performance parallel I/O. Each of these substrates can be combined with a compression layer, which currently supports zlib compression.

4. Evaluation

The design of OTF2 clearly focuses on providing a more interoperable, scalable and more functional tracing library. Nevertheless, the added features should not result in a slower or more memory consuming format. Therefore, we compared OTF2 with two other well-establish trace formats. The first is the EPILOG trace format [7] used by the Scalasca analysis tool [13]. The second is the Open Trace Format (OTF) [6] used by the Vampir/VampirTrace toolset [14]. Since the latter uses an internal memory buffer during runtime we used this for the comparisons.

However, comparing different trace data formats is a non-trivial task. In most cases the trace data libraries are closely connected with the according tracing tools. Therefore, it is necessary to decouple the trace data libraries from the tracing tools to eliminate all effects that originated in the tracing tools and not the data formats. In addition, different trace data formats do not store exactly the same information. They focus on different application behavior characteristics, different levels of detail, and provide a different amount of functionality. Furthermore, the results presented in this paper are measured with a first stable version of OTF2, which is not yet optimized for speed.

With this in mind, the comparisons presented in this paper are based on the following idea: Read an existing application trace file, select the comparable event records, and write only those records by using the different trace data libraries. With this method, identical data is written to the different libraries without generating timing information and random or even consecutive identifiers. So, the information sent to the libraries represents real application behavior but is exactly the same for all of them.

4.1. Runtime Memory Consumption

Memory consumption is one of the biggest issues when it comes to tracing, because if the tracing library runs out of memory, it must flush the trace to a file system. Those flushes usually need a long time, significantly perturbing the measurement. Thus, the comparison focuses on the memory consumption during measurement of an application, because less memory consumption leads to less flushes to the file system.

To compare the different libraries we took a set of benchmarks and applications:

- 104.milc ... 137.lu from SPEC MPI2007 benchmark[2]
- nas_pb_bt from NAS Parallel Benchmarks[3] (block tridiagonal solver),
- smg2000 - The SMG2000 Benchmark[4] (semicoarsening multigrid solver),
- sweep3d - ASCI SWEEP3D benchmark[5] (3D discrete neutron transport), and

[2]http://www.spec.org/mpi/
[3]http://www.nas.nasa.gov/Resources/Software/npb.html
[4]https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/
[5]http://www.ccs3.lanl.gov/pal/software/sweep3d/

- cosmo[6] - an atmospheric model code mainly developed by the DWD[7].

For runtime memory comparison, we used the VampirTrace memory storage model as a baseline. It reflects the memory consumption like it is achieved in practice, because the OTF library does not support buffer management. For the sake of completeness, we also added the results of a hypothetical test, where the Open Trace Format was used directly for the encoding of the memory buffer, not only for the storage on disk. Nevertheless, these results are only partially comparable, because it is not actually used during runtime. We did this to show that a further diminishment of the online memory consumption could not be achieved only by optimizing the library, but also required a redesign of the whole format. In first tests we have seen that OTF2 consumes about 70 % less memory than VampirTrace, about 15 % less memory space than EPILOG and about 20-35 % less space than OTF (see Fig. 4).

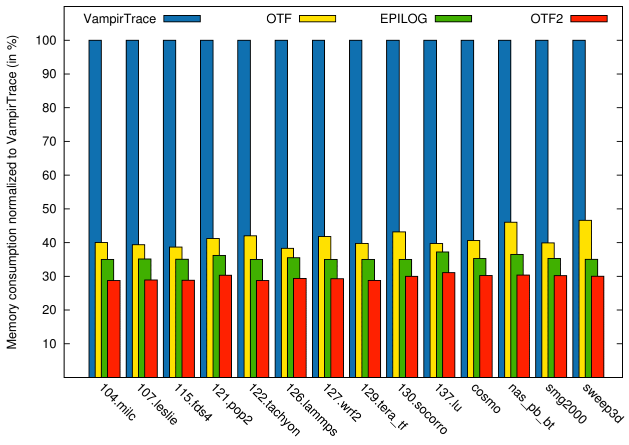

Figure 4. Memory usage in different applications. For details about the applications see Section 4.1.

4.2. Runtime Overhead

Measuring an application always introduces overhead into the application runtime, which can alter the application behavior and falsify the resulting trace. Therefore, even the tracing library must work very efficiently, because writing data with such a library into a memory buffer also directly influences the runtime overhead of the measurement. The two existing solutions, the EPILOG library and the buffering part of VampirTrace, are already very optimized in this direction. One of our goals for OTF2 was to introduce the on-line compression (see Section 3.2.2) without decreasing the speed of the tracing library.

We designed a benchmark for measuring the speed of all three solutions. It can read in an OTF2 trace and write the records into a memory buffer of OTF2, EPILOG, and VampirTrace. It is this way possible to simulate the measurement behavior of a wide range of applications. It does this for at least one second and counts how many event records could be written during this time. The benchmark is a parallel MPI program, to make the test case more realistic and to be able to read in large traces.

[6]http://www.cosmo-model.org/content/model/core/model/basicDesign.htm
[7]Deutscher Wetterdienst – German Meteorological Service

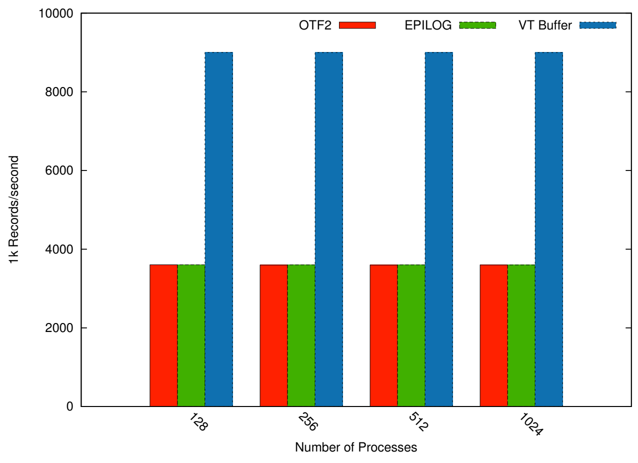

Figure 5. Events per Second.

The results of a first experiment with a trace of our Jacobi example can be seen in Figure 5. The Jacobi example implements a MPI variant of the Jacobi linear equation system solver. We did this particular test on a IBM BlueGene/P system, with 128, 256, 512, and 1024 processes, as it can be seen on the x-axis.

The results show that the EPILOG and OTF2 library are equally fast (≈3600k events/s). The VampirTrace buffer is about 2.5 times faster since all event attributes are aligned in memory, however, at the price of significantly increased memory requirements due to the padding, as was shown in the previous section. Since scalability in the time dimension is one of our main objectives, these tests indicate that OTF2 is a clear improvement over EPILOG in terms of memory usage without sacrificing performance.

5. Conclusion

In this paper we presented the Open Trace Format 2, a novel event trace data format with its associated read/write support library. It is designed for performance and scalability as well as better interoperability between different event trace analysis tools, in particular Scalasca and Vampir. Thus, it enables the user to analyze an application with both analysis tools without the need to run costly measurements multiple times. In addition, it is more memory efficient, offering the possibility to achieve measurements with less bias due to memory buffer flushes.

6. Future Work

The work on OTF2 is an ongoing process. One of the next steps is the further integration of the SION library [11] for other use cases then MPI programs, since SIONlib depends on the parallel programming paradigm used. With this, the next goal are test runs with large scale applications (>64k cores) to prove the scalability of the OTF2 concept. We also see further potential for reducing the runtime overhead of the online compression. In addition to that, there is an ongoing work to develop concepts and solutions to support upcoming new parallel paradigms such as PGAS or GPGPU applications. Furthermore, OTF2 is open to support other performance analysis tools to help reduce the barriers between them.

Acknowledgements

This work is funded by the German Federal Ministry of Education and Research in the SILC project.

References

[1] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proc. of SC2000: High Performance Networking and Computing*, Dallas, TX, USA, Nov 2000.

[2] Dieter Kranzlmüller, Michael Scarpa, and Jens Volkert. DeWiz - A Modular Tool Architecture for Parallel Program Analysis. In *Euro-Par 2003 Parallel Processing, Proc. 9th International Euro-Par Conference*, Springer LNCS 2790, pages 74–80, Klagenfurt, Austria, August 2003.

[3] S. Shende and A. D. Malony. The TAU Parallel Performance System, SAGE Publications. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[4] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain. *PARAVER Version 3.0 Trace-file Description*, June 2001.

[5] Intel Corp. Intel Trace Collector Reference Guide, Revision 8.0. Technical report, Intel Corp, 2010.

[6] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science ICCS 2006: 6th International Conference*, LNCS 3992, Reading, UK, May 2006. Springer.

[7] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[8] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *Proc. of International Parallel and Distributed Processing Symposium*, Mar 2007.

[9] Andreas Knüpfer and Wolfgang E. Nagel. Compressible memory data structures for event-based trace analysis. *Future Generation Computer Systems*, 22(3):359–368, Feb 2006.

[10] Thomas William, Hartmut Mix, and Bernd Mohr. Enhanced performance analysis of multi-core applications with an integrated tool-chain. In *International Conference on Parallel Computing, ParCo 2009*, September 2009.

[11] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel i/o to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[12] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P–A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany*, pages 1–12. Gauß-Allianz, Springer, June 2010.

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

[14] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool Set. In *Tools for High Performance Computing*, pages 139–155. Springer, July 2008.