Combining Instrumentation and Sampling for Trace-based Application Performance Analysis

Thomas Ilsche, Joseph Schuchart, Robert Schöne, and Daniel Hackenberg

Abstract Performance analysis is vital for optimizing the execution of high performance computing applications. Today different techniques for gathering, processing, and analyzing application performance data exist. Application level instrumentation for example is a powerful method that provides detailed insight into an application's behavior. However, it is difficult to predict the instrumentation-induced perturbation as it largely depends on the application and its input data. Thus, sampling is a viable alternative to instrumentation for gathering information about the execution of an application by recording its state at regular intervals. This method provides a statistical overview of the application execution and its overhead is more predictable than with instrumentation. Taking into account the specifics of these techniques, this paper makes the following contributions: (I) A comprehensive overview of existing techniques for application performance analysis. (II) A novel tracing approach that combines instrumentation and sampling to offer the benefits of complete information where needed with reduced perturbation. We provide examples using selected instrumentation and sampling methods to detail the advantage of such mixed information and discuss arising challenges and prospects of this approach.

1 Introduction

Performance analysis tools allow users to gain insight into the run-time behavior of applications and improve the efficient utilization of computational resources. Especially for complex parallel applications, the concurrent behavior of multiple tasks is

Technische Universität Dresden,
Center for Information Services and High Performance Computing (ZIH),
01062 Dresden, Germany
e-mail: {thomas.ilsche, joseph.schuchart, robert.schoene, daniel.hackenberg}@tu-dresden.de
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-16012-2_6

not always obvious, which makes the analysis of communication and synchronization primitives crucial to identify and eliminate performance bottlenecks.

Different techniques for conducting performance analyses have been established, each with their specific set of distinct advantages and shortcomings. These techniques differ in the type and amount of information they provide, e.g., about the behavior of one process or thread and the interaction between these parallel entities, the amount of data that is generated and stored, as well as the level of detail that is contained within the data. One contribution of this paper is to give a structured overview on these techniques to help users understand their nature. However, most of these approaches suffer from significant peculiarities or even profound disadvantages that limit their applicability for real-life performance optimization tasks:

- Full application instrumentation provides exhaustive information but comes with unpredictable program perturbation that can easily conceal the performance characteristics that need to be analyzed. Extensive event filtering may reduce the overhead, but this does require additional effort.
- Pure MPI instrumentation mostly comes with low overhead, but it provides only very limited information as the lack of application context for communication patterns complicates the performance analysis and optimization.
- Pure sampling approaches create very predictable program perturbation, but they lack communication and I/O information. Moreover, the classical combination with profiling for performance data presentation squanders important temporal correlations.
- Instrumentation-based approaches can only access performance counters at application events, thereby hiding potentially important information from in between these events.

A combination of techniques can often leverage the combined advantages and mitigate the weaknesses of individual approaches. We present such a combined approach that features low overhead and a high level of detail to significantly improve the usability and effectiveness of the performance analysis process.

2 Performance Analysis Techniques: Classification and Related Work

The process of performance analysis can be divided into three general steps: data acquisition, data recording, and data presentation [10]. These steps as well as common techniques for each step are depicted in Figure 1. Data acquisition reveals relevant performance information of the application execution for further processing and recording. This information is aggregated for storage in memory or persistent media in the data recording layer. The data presentation layer defines how the information is presented to the user to create insight for further optimization. In this section we present an overview of the often ambiguously used terminology and the state of the art of performance analysis tools.

2.1 Data Acquisition

2.1.1 Event-based instrumentation

Event-based instrumentation refers to a modification of the application execution in order to record and present certain intrinsic events of the execution, e.g., function entry and exit events. After the modification, these events trigger the data recording by the measurement environment at run-time. More specific events with additional semantics, such as communication or I/O operations, can often be derived from the execution of an API function.

The modification of the application can be applied on different levels. *Source code instrumentation* APIs used for a manual instrumentation, *source-to-source transformation* tools like PDT [14] and Opari [16], and *compiler instrumentation* require analysts to recompile the application under investigation after inserting instrumentation points manually or automatically. Thus, they can only be used for applications whose source code is available. Common ways to instrument applications without recompilation are *library wrapping* [5], *binary rewriting* (e.g., via DYNINST [3] or PEBIL [13]), and *virtual machines* [2].

All of these techniques are often referred to as *event-based instrumentation*, *direct instrumentation* [23], *event trigger* [11], *probe-based* measurement [17] or simply *instrumentation* and it is common to combine several of them in order to gather information on different aspects of an application run.

2.1.2 Sampling

Another common technique to obtain performance data is *sampling*, which describes the periodic interruption of a running program and inspection of its state. Sampling is realized by using timers (e.g., `setitimer`) or an overflow trigger of

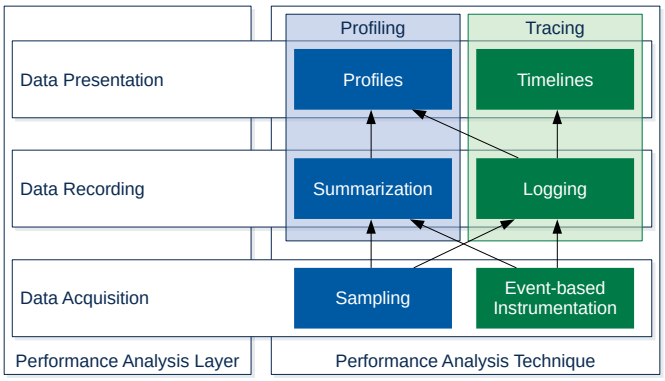

Fig. 1 Classification of performance analysis techniques (based on [11]). Valid combinations of techniques are connected with an arrow. Presenting data recorded by logging as a profile requires a post-processing summarization step.

hardware counters (e.g., using PAPI [6]). The most important aspects of inspecting the state of execution are the call-path and hardware performance counters. The call-path provides information about all functions (and regions) that are currently being executed. This information roughly corresponds to the enter/exit function events from event-based instrumentation. Additionally, the instruction pointer can be obtained, allowing sampling to narrow down hot-spots even within functions. However, the semantic interpretation of specific API calls is limited and can prevent the reconstruction of process interaction or I/O due to missing information. Moreover, the state of the application between two sampling points is unavailable for analysis.

In contrast to event-based instrumentation, sampling has a much more predictable overhead that mainly depends on the sampling rate rather than the event frequency. The user specifies the sampling rate and thereby controls the trade-off between measurement accuracy and overhead. While the complete information on specific events is not guaranteed with sampling, the recorded data can provide a statistical basis for analysis. For this reason, *sampling* is sometimes also referred to as *statistical sampling* or *profiling*.

2.2 Data Recording

2.2.1 Logging

Logging is the most elaborate technique for recording performance data. A timestamp is added to the information from the acquisition layer and all the information is retained in the recorded data. It can apply to both data from sampling and event-based instrumentation. Logging requires a substantial amount of memory and can cause perturbation and overhead during the measurement due to the I/O operations for writing a log-file to persistent storage. The term *tracing* is often used synonymously to *logging* and the data created by *logging* is a *trace*.

2.2.2 Summarization

By *summarizing* the information from the acquisition layer, the memory requirements and overhead of data recording are minimized at the cost of discarding the temporal context. For event-based instrumentation, values like sum of event duration, event count, or average message size can be recorded. Summarization of samples mainly involves counting how often a specific function is on the call-path, but performance metrics can also be summarized. This technique is also called *profiling*, because the data presentation of a summarized recording is a profile. A special hybrid case is the *phase* profile [15] or *time-series* profile [24], for which the information is summarized separately for successive phases (e.g., iterations) of the application. This provides some insight into the temporal behavior, but not to the extent of logging.

2.3 Data Presentation

2.3.1 Timelines

A timeline is a visual display of an application execution over time and represents the temporal relationship between events of a single or different parallel entities. This gives a detailed understanding of how the application is executed on a specific machine. In addition to the time dimension, the second dimension of the display can depict the call-path, parallel execution, or metric values. An example is given in Figure 2. Necessarily, timelines can only be created from logged data, not from summarized data.

2.3.2 Profiles

In a profile, the performance metrics are presented in a summary that is grouped by a factor such as the name of the function (or region). A typical profile is provided in Listing 1 and shows the distribution of the majority of time spent among functions. In such a *flat* profile the information is grouped by function name. It is also possible to group the information based on the call-path resulting in a *call-path* profile [24] (or *call graph* profile [8]). For performance metrics, the grouping can be done by metric or a combination of call-path and metric. Profiles can be created from either summarized data or logs.

2.4 Event Types

2.4.1 Code Regions

Several event types are of interest for application analysis. By far the most commonly used event types are code regions, which can be function calls either inside the application code or to a specific library, or more generally be any type of region such as loop bodies and other code structures. Therefore, code regions within the

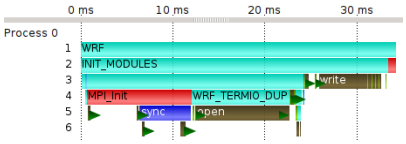

Fig. 2 A process timeline displaying the call-path and event annotations

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34     0.02     0.02     7208     0.00     0.00   open
 16.67     0.03     0.01      244     0.04     0.12   offtime
 16.67     0.04     0.01        8     1.25     1.25   memccpy
 16.67     0.05     0.01        7     1.43     1.43   write
```

Listing 1 Example output of `gprof` taken from its manual [19]

application are in the focus of this work. The knowledge of the execution time of an application function and its corresponding call-path is imperative for the analysis of application behavior. However, function calls can be extremely frequent and thus yield a high rate of trace events. This is especially true for C++ applications, where short methods are very common, making it difficult to keep the run-time overhead of instrumentation and tracing low.

2.4.2 Communication and I/O Operations

The exchange of data between tasks (communication) is essential for parallel applications and highly influential on the overall performance. Communication events can contain information about the sender/receiver, message size, and further context such as MPI tags. File I/O is a form of data transfer between a task and persistent storage. It is another important aspect for application performance. Typical file I/O events include information about the active task, direction (read/write), size, and file name.

2.4.3 Performance Metrics

The recording of the above mentioned events only gives limited information on the usage efficiency of shared and exclusive resources. Additional metrics describing the utilization of these resources are therefore important performance measures. The set of metrics consists of (but is not limited to) hardware performance counter (as provided by PAPI), operating system metrics (e.g., via rusage), and energy and power measurements.

2.4.4 Task Management

The management of tasks (processes and threads) is also of interest for application developers. This set of events includes task creation (fork), shutdown (join), and the mapping from application tasks to OS threads.

2.5 Established Performance Analysis Tools

Several tools support the different techniques mentioned in Section 2 and in parts combine some of them.

The Scalasca [7] package focuses on displaying profiles, but logged data is used for a special post-processing analysis step. VampirTrace [18] mainly focuses on refined tracing techniques but comes with a basic profiling mode and external tools for extracting profile information from trace data. These two software packages rely

mostly on different methods of event-based instrumentation. The Tuning and Analysis Utilities (TAU) [22] implement a measurement system specialized for profiling with some functionality for tracing. TAU supports a wide range of instrumentation methods but a hybrid mode that uses call-path sampling in combination with instrumentation is also possible [17]. The performance measurement infrastructure Score-P [12] has both sophisticated tracing and profiling capabilities. It mainly acquires data from event-based instrumentation, but recent work [23] introduced call-path sampling for profiling. The graphical tool Vampir [18] can visualize traces created with Score-P, VampirTrace or TAU in the form of timelines or profiles. Similar to the above mentioned, the Extrae software records traces based on various instrumentation mechanisms. Sampling in Extrae is supported by interval timers and hardware performance counter overflow triggers. The sampling data of multiple executions of a single code region can be combined into a single detailed view using folding [21]. This combined approach provides increased information about repetitive code regions. HPCToolkit [1] implements sampling based performance recording. It provides sophisticated techniques for stack unwinding and call-path profiling. The data can also be recorded in a trace and displayed in a timeline trace viewer. All previously mentioned tools have a strong HPC background and are therefore designed to analyze large scale programs. For example Scalasca and VampirTrace/Vampir can handle applications running on more than 200,000 cores [25, 9].

Similar combinations of techniques can also be seen in tools without a specialization for HPC. The Linux' perf infrastructure [4] consists of a user space tool and a kernel part that allows for application-specific and system-wide sampling based on both hardware events and events related to the operating system itself. Support for instrumentation-based analysis is added through kprobes, uprobes, and tracepoint events. The infrastructure part of perf is also used by many other tools as it provides the basis to read hardware performance counters on Linux with PAPI. The GNU profiler (gprof) [8] provides a statistical profile of function run-times, but also employs instrumentation by the compiler to derive accurate number-of-calls figures.

3 Combining Multiple Performance Analysis Techniques: Concept and Experiences

As discussed in Section 2, sampling and event-based instrumentation have different strengths and weaknesses. A combined performance analysis approach can use instrumentation for aspects of the application execution for which full information is desired and sampling to complement the performance information with limited perturbation. We discuss two new approaches and evaluate them based on prototype implementations for the VampirTrace plugin counter interface [20]: (I) Instrumenting MPI calls and sampling call-paths; and (II) Instrumenting application regions but sampling hardware performance counters.

3.1 MPI Instrumentation and Call-Path Sampling

Performance analysis of parallel applications is often centered around messages and synchronization between processes. In the case of applications using MPI, it is common practice to instrument the API calls to get information about every message during application execution [12, 22, 7, 18]. The MPI profiling interface (PMPI) allows for a convenient and reliable instrumentation that only requires re-linking and can even be done dynamically when using shared libraries. Using sampling for message passing information would significantly limit the analysis, e.g., since reliable message matching requires information about each message. However, only recording message events lacks context for a holistic analysis, as for example the root cause of inefficient communication or load imbalances cannot be determined. Call-path sampling is a viable option to complement message recording, as it provides rich context information but–unlike compiler instrumentation–does not require recompilation. The projected run-time perturbation and overhead of this approach is very promising: On the one hand, the overhead can be controlled by adjusting the sampling rate. On the other hand, MPI calls for communication can be assumed to have a certain minimum run-time, thereby limiting the event frequency as well as the overhead caused by this instrumentation. Some applications that make excessive use of many small messages, especially when using non-blocking MPI functions, are still difficult to analyze efficiently with this approach, but this also applies to MPI only instrumentation.

3.1.1 Implementation

We implemented a prototypical sampling support for VampirTrace as a plugin. Whenever VampirTrace registers a task for performance analysis, the plugin is activated and initializes a performance counter based interrupt, e.g., every 1 million cycles. Whenever such a counter overflow occurs, the plugin checks whether the current functions on the stack belong to the main application, i.e., are not part of a library, and adds function events for all functions on the call-path. MPI library calls and communication events are recorded using the instrumented MPI library of VampirTrace. The application does not have to be recompiled to create a trace.

3.1.2 Results

Figure 3 shows the visualization of a trace using an unmodified version of Vampir [18], i.e., without specific support for sampled events. The MPI function calls and messages are clearly visible due to the instrumented MPI library. The application functions, and thus the context of the communication operation, are visible as samples. This already allows users to analyze the communication, possible bottlenecks, and imbalances. Containing the complete call stack in the trace remains as future work.

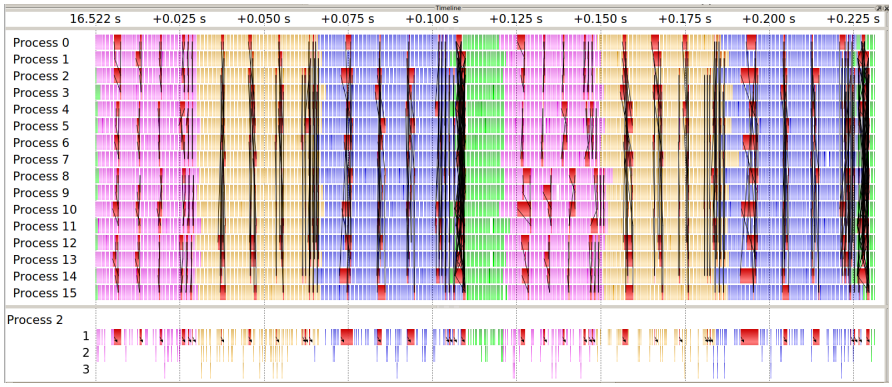

Fig. 3 Vampir visualization of a trace of the NPB BT MPI benchmark created using an instrumented MPI library (MPI functions displayed red and messages as black lines) and sampling for application functions (`x_solve` colored pink, `y_solve` yellow, `z_solve` blue). Stack view of one process shown below the master timeline.

Figure 4 shows the measured overhead for recording traces of the analyzed NPB benchmark. The overhead is very high for the fully instrumented version, while sampling application functions in addition to the instrumented MPI library only adds a marginal overhead. Thus, while providing all necessary information on communication events and still allowing the analysis of the application's call-paths, the overhead can be decreased significantly. These results demonstrate the advantage of combining call-path sampling and library instrumentation.

3.2 Sampling Hardware Counters and Instrumenting Function Calls and MPI Messages

As a second example, we demonstrate the sampling of hardware counter values while tracing function calls and MPI events with traditional instrumentation. In contrast to the traditional approach of recording hardware counter values on every application event, this approach has two important advantages: First, in long running code regions with filtered or no subroutine calls, the sampling approach still provides intermediate data points that allow users to estimate the application performance for smaller parts of this region. Second, for very short code regions, the overhead of the traditional approach can cause significant program perturbation and recorded performance data that does not necessarily contain valuable information for the optimization process. Moreover, reading hardware counter values in short running functions can cause misleading results due to measurement perturbation.

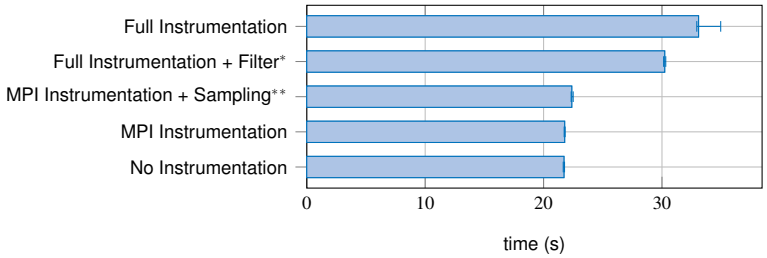

Fig. 4 Run-time of different performance measurement methods for NPB BT CLASS B, SIZE 16 on a dual socket Sandy Bridge system. Median of 10 repeated runs with minimum/maximum bars. * Filtered functions: `matmul_sub`, `matvec_sub`, `binvrhs`, `binvcrhs`, `lhsinit`, `exact_solution`; ** Sampling rate of 2.6 kSa/s.

3.2.1 Implementation

For each application thread, the plugin creates a monitoring thread that wakes up in in certain intervals intervals to query and record the hardware counters and sleeps the rest of the time.

3.2.2 Results

Figure 5 shows the visualization of a trace of NPB FT that was acquired using compiler instrumentation and an instrumented MPI library. The trace contains two different versions of the same counter (retired instructions), one recorded on every enter/exit event (middle part) and the second sampled every 1 ms (bottom). On the one hand, the instrumented counter shows peaks in regions with a high event rate

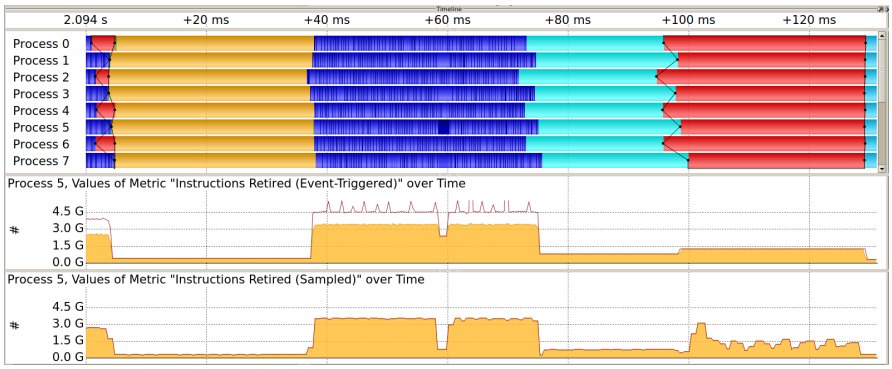

Fig. 5 Vampir visualization of a trace of the NPB FT benchmark acquired through compiler instrumentation and instrumented MPI library (master timeline, top) including an event-triggered (middle) and a sampled (bottom) counter for retired instructions. Colors: MPI red, FFT blue, `evolve` yellow, transpose light blue

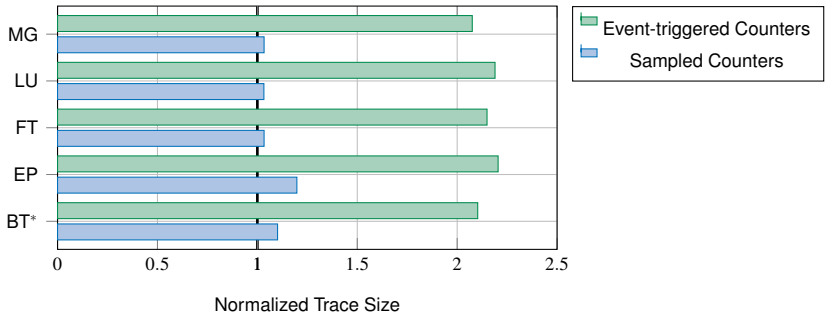

Fig. 6 Normalized trace sizes of NPB CLASS B benchmarks containing hardware performance counters either triggered by instrumentation events or asynchronously sampled (1 kSa/s). Baseline: trace without counters. * Filtered functions: `matmul_sub`, `matvec_sub`, `binvcrhs`, `exact_solution`.

due to very short-running functions. This large amount of information is usually of limited use except for analyzing these specific function calls. The sampled counter does not provide this wealth of information but still reflects the average application performance in these regions correctly. On the other hand, the sampled counter provides additional information for long running regions, e.g., MPI functions and the `evolve_` function. This information is useful for having a more fine-grained estimation of the hardware resource usage of these code areas. Furthermore, Figure 6 demonstrates that sampling counter values can be used to significantly reduce trace sizes compared to recording counter values through instrumentation. After all, combining the approaches outlined in this section and in Section 3.1 is feasible and will remain as future work.

4 Conclusions and Future Work

In this paper, we presented a comprehensive overview of existing performance analysis techniques and the tools employing them, taking into account their specific advantages and disadvantages. In addition, we discussed the general approach of combining the existing techniques of instrumentation and sampling to leverage each of their potential. We demonstrated this with two practical examples, showing results of prototype implementations for (I) sampling application function call-paths while instrumenting MPI library calls; and (II) sampling hardware performance counter values in addition to traditional application instrumentation. The results confirm that this combined approach has unique advantages over the individual techniques.

Based on the work presented here, we will continue to explore ways of combining instrumentation and sampling for performance analysis by integrating and extending open-source tools available for both strategies. Taking more event types into consideration is another important aspect. For instance, I/O operations and CUDA

API calls are viable targets for instrumentation while resource usage (e.g. memory) can be sampled.

Another interesting aspect is the visualization of traces based on call-path samples in a close-up view. It is challenging to present this non-continuous information in an intuitively understandable fashion. We will also further investigate the scalability of our combined approach. The effects of asynchronously sampling in large scale systems that require a very low OS noise to operate efficiently needs to be studied. Our goal is a seamless integration of instrumentation and sampling for gathering trace data to be used in a scalable and holistic performance analysis technique.

Acknowledgements This work has been funded by the Bundesministerium für Bildung und Forschung via the research project CoolSilicon (BMBF 16N10186) and the Deutsche Forschungsgemeinschaft (DFG) via the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing" (HAEC, SFB 921/1 2011). The authors would like to thank Michael Werner for his support.

References

1. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6), 2010.
2. Walter Binder. Portable and accurate sampling profiling for java. *Software: Practice and Experience*, 36(6), 2006.
3. Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14, 2000.
4. Arnaldo Carvalho de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, 2010.
5. Robert Dietrich, Thomas Ilsche, and Guido Juckeland. Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures. *International Conference on Parallel Processing Workshops*, 2010.
6. Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03. Springer, 2003.
7. 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), April 2010.
8. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, 1982.
9. Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E. Nagel, and Stephen Poole. Optimizing i/o forwarding techniques for extreme-scale event tracing. *Cluster Computing*, June 2013.
10. R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.
11. Guido Juckeland. *Trace-based Performance Analysis for Hardware Accelerators*. PhD thesis, TU Dresden, 2012.
12. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*. Springer, September 2012.

13. M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.
14. Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Forschungszentrum Juelich, Reid Rivenburgh, Craig Rasmussen, and Bernd Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the international conference on Supercomputing*. IEEE, 2000.
15. Allen D. Malony, Sameer S. Shende, Alan Morris, G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata. Phase-based parallel performance profiling. In *In Proceedings of the PARCO 2005 conference*, 2005.
16. Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Towards a performance tool interface for openmp: An approach based on directive rewriting. In *In Proceedings of the Third Workshop on OpenMP (IWOMP*, 2001.
17. Alan Morris, Allen D. Malony, Sameer Shende, and Kevin A. Huck. Design and implementation of a hybrid parallel performance measurement system. In *ICPP*, pages 492–501, 2010.
18. Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15. IOS Press, 2008.
19. Jeffrey Osier. The GNU gprof manual (info gprof).
20. Robert Schöne, Ronny Tschüter, Thomas Ilsche, and Daniel Hackenberg. The vampirtrace plugin counter interface: Introduction and examples. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*. Springer, 2011.
21. Harald Servat, Germán Llort, Judit Giménez, Kevin Huck, and Jesús Labarta. Folding: Detailed analysis with coarse sampling. In *Tools for High Performance Computing 2011*. Springer, 2012.
22. Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2), 2006.
23. Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Felix Wolf, and Brian J. N. Wylie. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *Proc. of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2011.
24. Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the international conference on Supercomputing*. ACM, November 2009.
25. Brian J. N. Wylie, Markus Geimer, Bernd Mohr, David Böhme, Zoltán Szebenyi, and Felix Wolf. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters*, 20(4):397–414, December 2010.