Reducing the overhead of direct application instrumentation using prior static analysis*

Jan Mußler[1], Daniel Lorenz[1], and Felix Wolf[1,2,3]

[1] Jülich Supercomputing Centre, 52425 Jülich, Germany
[2] German Research School for Simulation Sciences, 52062 Aachen, Germany
[3] RWTH Aachen University, 52056 Aachen, Germany

Abstract. Preparing performance measurements of HPC applications is usually a tradeoff between accuracy and granularity of the measured data. When using direct instrumentation, that is, the insertion of extra code around performance-relevant functions, the measurement overhead increases with the rate at which these functions are visited. If applied indiscriminately, the measurement dilation can even be prohibitive. In this paper, we show how static code analysis in combination with binary rewriting can help eliminate unnecessary instrumentation points based on configurable filter rules. In contrast to earlier approaches, our technique does not rely on dynamic information, making extra runs prior to the actual measurement dispensable. Moreover, the rules can be applied and modified without re-compilation. We evaluate filter rules designed for the analysis of computation and communication performance and show that in most cases the measurement dilation can be reduced to a few percent while still retaining significant detail.

1 Introduction

The complexity of high-performance computing applications is rising to new levels. In the wake of this trend, not only the extent of their code base but also their demand for computing power is rapidly expanding. System manufacturers are creating more powerful systems to deliver the necessary compute performance. Software tools are being developed to assist application scientists in harnessing these resources efficiently and to cope with program complexity. To optimize an application for a given architecture, different performance-analysis tools are available, utilizing a wide range of performance-measurement methodologies [17,13,21,18,7]. Many performance tools used in practice today rely on *direct instrumentation* to record relevant events, from which performance-data structures such as profiles or traces are generated. In contrast to statistical sampling, direct instrumentation installs calls to measurement routines, so-called hooks, at function entry and exit points or around call sites. This can be done on multiple levels ranging from the source code to the binary file or even the

* This material is based upon work supported by the US Department of Energy under Award Number DE-SC0001621.

Published in "Proc. of Europar 2011", LNCS 6852, pp. 65–76, Springer, 2011.
The original publication is available at www.springerlink.com

memory image [20]. Often the compiler can inject these hooks automatically using a profiling interface specifically designed for this purpose.

Of course, instrumentation causes measurement intrusion – not only dilating the overall runtime and prolonging resource usage but also obscuring measurement results – especially, if the measurement overhead is substantial. If applied indiscriminately, the measurement dilation can render the results even useless. This happens in particular in the presence of short but frequently-called functions prevalent in C++ codes. In general, the measurement overhead increases with the rate at which instrumentation points are visited. However, depending on the analysis objective, not all functions are of equal interest and some may even be excluded from measurement without loosing relevant detail. For example, since the analysis of message volumes primarily focuses on MPI routines and their callers, purely local computations may be dispensable. Unfortunately, manually identifying and instrumenting only relevant functions is no satisfactory option for large programs. Although some automatic instrumentation tools [6,21] offer the option of explicitly excluding or including certain functions to narrow the measurement focus, the specification of black and white lists usually comes at the expense of extra measurements to determine suitable candidates.

To facilitate low-overhead measurements of relevant functions without the need for additional measurement runs, we employ static analysis to automatically identify suitable instrumentation candidates based on structural properties of the program. The identification process, which is accomplished via binary inspection using the the Dyninst library [3], follows filter rules that can be configured by refining and combining several base criteria suited for complementary analysis objectives. The resulting instrumentation specification is then immediately applied to the executable via binary re-writing [22], eliminating the need for re-compilation. Our methodology is available in the form of a flexible standalone instrumentation tool that can be configured to meet the needs of various applications and performance analyzers. Our approach significantly reduces the time-consuming work of filter creation and improves the measurement accuracy by lowering intrusion to a minimum. An evaluation of different filter criteria shows that in most cases the overhead can be reduced to only a few percent.

Our paper is structured as follows: After reviewing related work in Section 2, we present the design of the configurable instrumentation tool in Section 3. Then, in Section 4, we discuss the base filter criteria and the heuristics involved in their implementation. A comprehensive experimental evaluation of these criteria in terms the number of instrumented functions and the resulting measurement dilation is given in Section 5. Finally, we draw conclusions and outline future work in Section 6.

2 Related Work

To generally avoid the overhead of direct instrumentation, some tools such as HPCToolkit [13] resort to sampling. Although researchers recently also started combining sampling with direct instrumentation [19], the choice between the two

options is usually a trade-off between the desired expressiveness of the performance data and unwanted measurement dilation. Whereas sampling allows the latter to be controlled with ease, just by adjusting the sampling frequency, it delivers only an incomplete picture, potentially missing critical events or providing inaccurate estimates. Moreover, accessing details of the program state during the timer interrupt, such as arguments of the currently executed function, is technically challenging. Both disadvantages together make direct instrumentation the favorite method for capturing certain communication metrics such as the size of message payloads. This insight is also reflected in the current design of the MPI profiling interface [14], whose interposition-wrapper concept leverages direct instrumentation. However, to avoid excessive runtime overhead, the number of direct instrumentation points need to be selected with care, a task for which our approach now offers a convenient solution. If only the frequency of call-path visits is of interest, also optimizations such as those used by Ball and Larus for path profiling can be chosen [2].

Among the tools that rely on direct instrumentation, the provision of black lists to exclude functions from instrumentation (or white lists to include only a specific subset) is the standard practice of overhead minimization. In Scalasca [7] and TAU [21], the specification of such lists is supported through utilities that examine performance data from previous runs taken under full instrumentation. Selection criteria include the ratio between a function's execution time and its number of invocations or whether the function calls MPI – directly or indirectly. Yet, in malign cases where the overhead of full instrumentation is excessive, the required extra run may be hard or even impossible to complete in the first place. The selection lists are applied either statically or dynamically. The latter is the preferred method in combination with compiler instrumentation, which can be configured only at the granularity of entire files. In addition to user-supplied filter lists, TAU provides a runtime mechanism called *throttling* to automatically disable the instrumentation of short but frequently executed functions. A general disadvantage of runtime selection, whether via filter lists or automatically, however, is the residual overhead caused by the dynamic inspection of function identifiers upon each function call. Our solution, in contrast, neither requires any extra runs nor performs any dynamic checks.

Another generic instrumenter was designed by Geimer et al. [6]. Like ours, it can be configured to support arbitrary measurement APIs. Whereas we analyze and modify the binary, their instrumenter identifies potential instrumentation points in the source code. While allowing the restriction of target locations according to file and function names, the lack of static source-code analysis functionality prevents it from providing suggestions as to which functions should be instrumented. Moreover, changing the instrumentation always entails an expensive re-compilation.

An early automatic filter mechanism was developed as an extension of the OpenUH compiler's profiling interface [8]. Here, the compiler scores functions according to their estimated number of executions and their estimated duration, which are derived from the location of their call sites and their number

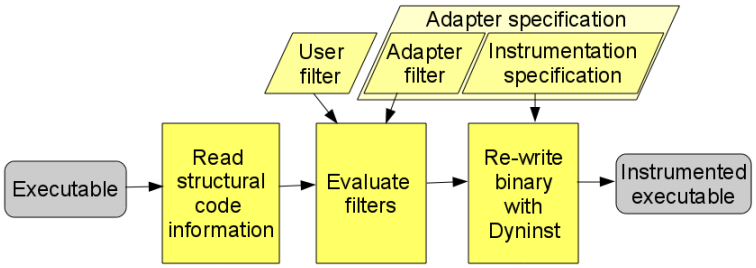

Fig. 1. The basic instrumentation workflow.

of instructions, respectively. Based on this assessment, the compiler skips the instrumentation of those functions that are either short or called within nested loops. However, generally not instrumenting small functions was criticized by Adhianto et al.[1]. They argue that small functions often play a significant role, for example, if they include synchronization calls important to parallel performance. In our approach, providing rules that explicitly define exceptions, for example, by forcing the instrumenter to include all functions that call a certain synchronization primitive, can avoid or mitigate the risk of missing critical events.

If measurement overhead cannot be avoided without sacrificing analysis objectives, overhead compensation offers an instrument to retroactively improve the accuracy of the measured data. Initially developed for serial applications [10], it was later extended to account for overhead propagation in parallel applications [11]. The approach is based on the idea that every call to the measurement system incurs a roughly constant and measurable overhead with a deterministic and reversible effect on the performance data. However, variations in memory or cache utilization may invalidate this assumption to some degree.

3 A Configurable Instrumenter

Figure 1 illustrates the different steps involved in instrumenting an application and highlights the functional components of our instrumenter. As input serves a potentially optimized application executable, which is transformed into a ready-to-run instrumented executable, following the instructions embodied by user filters and adapter specifications. The instrumentation process starts with the extraction of structural information from the binary program, a feature supported by the Dyninst API. Although the inclusion of debug information into the executable during compilation is not mandatory, it tends to enrich the available structural information and can help formulate more sophisticated filter rules. As a next step, the instrumenter parses the provided filter specifications and determines the functions to be instrumented. Optionally, the instrumenter can print the names of instrumentable functions to simplify the creation of filter lists. The instrumentation itself is applied using Dyninst's binary rewriting capabilities.

The raw instrumenter can be configured in two different ways: First, developers of performance tools can provide an adapter specification (top right in

Figure 1) to customize the instrumenter to their tool's needs. This customization includes the specification of code snippets such as calls to a proprietary measurement API to be inserted at instrumentation points. In addition, the adapter may include a predefined filter that reflects the tool's specific focus. Second, application developers can augment this predefined filter by specifying a user filter (top center in Figure 1) to satisfy application- or analysis-specific requirements. Below, we explain these two configuration options in detail.

3.1 Adapter Specification

The adapter specification is intended to be shipped together with a performance tool. It consists of a single XML document, which is both human readable (and editable) and at the same time accessible to automatic processing through our instrumenter. The format provides four different element types:

- The description of additional dependencies, for example, to measurement libraries that must be linked to the binary.
- The definition of optional adapter filter rules. These adhere to the same syntax as the user filter rules, which are introduced further below. The filter rules allow the exclusion of certain functions such as those belonging to the measurement library itself or those known to result in erroneous behavior. For example, when using Scalasca, which requires the measurement library to be statically linked to the application, the adapter filter would prevent the library itself from being instrumented.
- The definition of global variables.
- The definition of a set of code snippets to be inserted at instrumentation points.

The instrumenter supports instrumentation on three different levels: (i) function, (ii) loop, and (iii) call site. There are up to four instrumentation points associated with each level, plus two for initialization and finalization:

- *before*: Immediately before a call site (i.e., function call) or loop.
- *enter*: At function entry or at the start of a loop iteration.
- *exit*: At function exit or at the end of a loop iteration.
- *after*: Immediately after a call site (i.e., function call) or loop.
- *initialize*: Initialization code which is executed once for each instrumented function, call site, or loop.
- *finalize*: Finalization code which is executed once for each instrumented function, call site and loop.

The initialization and finalization code is needed, for example, to register a function with the measurement library or to release any associated resources once they are no longer in service.

To access an instrumentation point's context from within the inserted code, such as the name of the enclosing function or the name of the function's source file, the instrumenter features a set of variables in analogy to [6]. These variables

```
<adapter>
  <functions>
    <variables><var name="i" size="4" /></variables>
    <init>i = 0;</init>
    <enter>i = i + 1;
           printf("entering %s for the %d time\n",@ROUTINE@,i);
    </enter>
  </functions>
</adapter>
```

Fig. 2. Example adapter specification that counts the number of visits to an instrumented function and during each invocation prints a message which contains the function name and the accumulated number of visits.

are enclosed by `@` and include, among others, the following items: `ROUTINE`, `FILE`, and `LINE`. To concatenate strings, we further added the `.` operator. At instrumentation time, a single `const char*` string will be generated from the combined string. In addition to specifying default code snippets to be inserted at the six locations listed above, an adapter specification may define uniquely named alternatives, which can be referenced in filter rules to tailor the instrumentation to the needs of specific groups of functions. An example adapter specification is shown in Figure 2. So far, we created adapter specifications for Scalasca, TAU and the Score-P measurement API [15]. The latter is a new measurement infrastructure intended to replace the proprietary infrastructures of several production performance tools.

3.2 Filter Specification

While the adapter customizes the instrumenter for a specific tool, the user filter allows the instrumenter to be configured for a specific application and/or analysis objective such as communication or computation. It does so by restricting the set of active instrumentation points of the target application.

The filter is composed of *include* and *exclude* elements, which are evaluated in the specified order. The *exclude* elements remove functions from the set, whereas *include* elements add functions to the set. A filter element consist of a particular set of properties a function must satisfy. The properties can be combined through the use of the logical operators *and*, *or*, and *not*. The properties are instances of base filter criteria, which are described in Section 4. For each instrumentable function, every rule is evaluated to decide whether the function matches the rule or not. An example for a filter definition is given in Figure 3. In addition to defining whether a function is instrumented or not, the user can also change the default code to be inserted by selecting alternative code snippets from the adapter specification, referencing them by the unique name that has been assigned there. Separate snippets can be chosen depending on whether the instrumentation occurs around functions, call sites, or loops.

```
<filter name="mpicallpath" instrument="functions=function" start="none">
<include>
   <property name="path">
       <functionnames match="prefix">MPI mpi</functionnames>
   </property>
</include>
</filter>
```

Fig. 3. Example for a filter definition that instruments all functions that appear on a call path leading to an MPI function.

4 Filter Criteria

The purpose of our filter mechanism is to exclude functions that either lie outside our analysis objectives or whose excessive overhead would obscure measurement results. To avoid instrumenting any undesired functions, the instrumenter supports selection criteria (i) based on string matching and/or (ii) based on the program structure. String matching criteria demand that a string (e.g., the function name) has a certain prefix or suffix, contains a certain substring, or matches another string completely. String matching can be applied to function names, class names, namespaces, or file names. It is a convenient method, for example, to prevent the instrumentation of certain library routines that all start with the same prefix. In contrast, structural criteria take structural properties of a given function into account.

The first group of structural properties considers a function's position in the call tree, that is, its external relationships to other functions. This is useful to identify functions that belong to the context of functions in the center of our interest. For example, if the focus of the analysis are MPI messaging statistics, the user might want to know from where messaging routines are called but at the same time may afford to skip purely local computations in the interest of improved measurement accuracy.

Call path: Checks whether a function may appear on the call path to a specified set of functions, for example, whether the function issues MPI calls – either directly or indirectly. Unfortunately, since the decision is based on prior static analysis of the code, virtual functions or function pointers are ignored.

Depth: Checks whether a function can be called within a certain call-chain depth from a given set of functions. Relying on static call-graph analysis, as well, this property suffers from the same restrictions as the call-path property.

The second group of structural properties considers indicators of a function's internal complexity. This is motivated by short but frequently called functions that often contribute little to the overall execution time but cause overproportional overhead.

Lines of code: Checks whether the number of source lines of a function falls within a given range. Using available debug information, the instrumenter

computes the number of source lines between the first entry point and the last exit point of a function. Note that the number of source lines may depend on the length of comments or the coding style. Moreover, inlining of macros or compiler optimizations may enlarge the binary function compared to its source representation.

Cyclomatic complexity: Checks whether the cyclomatic complexity [12] of a function falls within a given range. The cyclomatic complexity is the number of linearly independent paths through the function. We chose the variant that takes also the number of exit points into account. It is defined as $E - N + P$, with N representing the number of nodes in the control flow graph (i.e., the number of basic blocks), E the number of edges between these blocks, and P the number of connected components in the graph, which is 1 for a function. Again, inlining and compiler optimizations may increase the cyclomatic complexity in comparison to what a programmer would expect.

Number of instructions: Checks whether the number of instructions falls within a given range. Since the number of instructions is highly architecture and compiler dependent, it is challenging to formulate reasonable expectations.

Number of call sites: Checks whether the function contains at least a given number of function calls. Note that the mere occurrence of a call site does neither imply that the function is actually called nor does it tell how often it is called.

Has loops: Checks whether a function contains loops. Here, similar restrictions apply as with the number of call sites.

Of course, it is also possible to combine these criteria, for example, to instrument functions that either exceed a certain cyclomatic complexity threshold or appear on a call path to an MPI function.

5 Evaluation

In this section we evaluate the effectiveness of selected filter rules in terms of the overhead reduction achieved and the loss of information suffered (i.e., the number of functions not instrumented). The latter, however, has to be interpreted with care, as not all functions may equally contribute to the analysis goals. Since all of our filter criteria are parameterized, the space of filter rules that could be evaluated is infinitely large. Due to space and time constraints, we therefore concentrated on those instances that according to our experiences are the most useful ones. They are are listed below. Criteria not considered here will be the subject of future studies.

- MPI Path: Instrument only functions that appear on a call path to an MPI function. This filter allows the costs of MPI communication to be broken down by call path, often a prerequisite for effective communication tuning.
- CC 2+: Instrument all functions that have at least a cyclomatic complexity of two. This filter removes all functions that have only one possible path of execution.

Application	Language	Full	CC 2+	CC 3+	LoC 5+	MPI Path
104.milc	C	261	51%	38%	71%	43%
107.leslie3d	Fortran	32	78%	66%	75%	28%
113.GemsFDTD	Fortran	197	62%	58%	81%	12%
115.fds4	C/Fortran	238	80%	74%	88%	0.4%
121.pop2	C/Fortran	982	65%	53%	77%	16%
122.tachyon	C	342	35%	27%	61%	5%
125.RAxML	C	313	77%	65%	85%	25%
126.lammps	C++	1378	56%	43%	64%	39%
128.GAPgeomfem	C/Fortran	36	61%	53%	72%	31%
130.socorro	C/Fortran	1331	50%	41%	74%	24%
132.zeusmp2	C/Fortran	155	84%	80%	89%	46%
137.lu	Fortran	35	66%	60%	77%	43%
Cactus Carpet	C++	3539	35%	29%	50%	6%
Gadget	C	402	62%	52%	26%	21%

Table 1. Number of functions instrumented under full instrumentation and percentage of functions instrumented after applying different filter rules.

- CC 3+: Instruments all functions that have a cyclomatic complexity of three or higher. Compared to the CC 2+ filter, functions need an additional loop or branch not to be removed.
- LoC 5+: Instrument all functions with five or more lines of code. This filter is expected to remove wrapper functions as well as getters, setters, or other one liners.

As test cases, we selected the SPEC MPI2007 benchmark suite [16], a collection of twelve MPI applications from various fields with very different characteristics; Gadget [5], a simulation that computes the collision of two star clusters; and Cactus Carpet [4], an adaptive mesh refinement and multi-patch driver for the Cactus framework. A full list of all applications can be found in Table 1 together with information on their programming language. We built all applications using the GNU compiler with optimization level `O2` enabled. All measurements were performed on Juropa [9], an Intel cluster at the Jülich Supercomputing Centre. Our instrumenter was linked to version 6.1 of Dyninst. To improve interoperability with the GNU compiler, we added GCC exception handling functions to the list of non-returning functions in Dyninst. We ran our test cases using the Scalasca measurement library in profiling mode, which instruments all MPI function by default through interposition wrappers.

Table 1 lists the number of instrumented functions when applying different filter rules including full instrumentation. The numbers do not include MPI functions, which are always instrumented. Also, the instrumenter was configured not to instrument the Scalasca measurement system itself. Otherwise, all functions Dyninst identified in the binary, which do not include dynamically linked libraries, were potential instrumentation candidates. The number of functions varies greatly among the fourteen applications, with *107.leslie3d* having only 32 compared to the two C++ codes with 1378 and 3539 functions, respectively. Judging by the fraction of eliminated functions, the MPI Path filter seems to

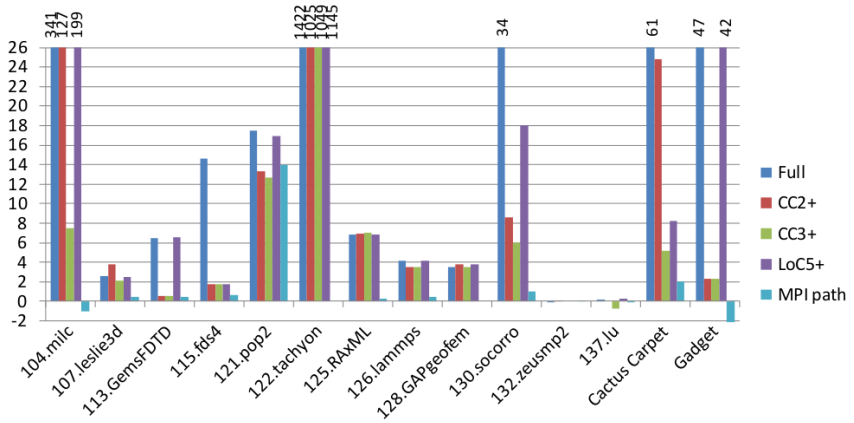

Fig. 4. Runtime overhead of the fully instrumented binary and after applying different filters. The values are given in percent compared to an uninstrumented run. Values exceeding 26% are clipped. Missing bars imply zero overhead. In general, measurement inaccuracies prevented a precise representation of values around zero, sometimes resulting in negative overhead figures.

be the most aggressive one, in one instance (*115.fds4*) leaving only the main function instrumented. The LoC 5+ filter, by contrast, leads only to relatively mild eliminations, with most codes loosing less then 40%. Finally, the difference between the two cyclomatic filters, which together occupy a solid middle ground in terms of their aggressiveness, is significant but not too pronounced.

The measurement overheads observed for the individual combinations of applications and filters are presented in Figure 4. Seven of the fourteen applications show less than 8% overhead even under full instrumentation, indicating that full instrumentation is not generally impracticable. Among the remaining applications, three including the C++ code Cactus Carpet exhibit extreme overheads above 50% without filters. The worst case is clearly *122.tachyon* with more than 1,000% overhead, which, however, contains functions with only two binary instructions. In almost all cases, with the exception of *121.pop2*, at least one filter exists that was able to reduce the overhead to 2% or less – within the limits of our measurement accuracy. For *121.pop2*, we achieved only a moderate reduction, although with 13% the lowest overhead of *121.pop2* was not alarming. As a general trend, the MPI Path filter resulted in the lowest overhead. Again, the only exception is 121.pop2, where many functions contain error handling code that may call MPI functions such as MPI_Finalize and MPI_Barrier. Thus, many functions were instrumented that actually do not call MPI during a normal run. Whereas the LoC 5+ filter did often enough fail to yield the desired overhead decrease, CC 3+ can be seen as a good compromise, often with higher although still acceptable overhead below 10% – but on the other hand with less functions removed from instrumentation and, thus, with less loss of information.

Finally, the ratio between the fraction of filtered functions and the overhead reduction can serve as an indicator of how effective a filter is in selecting functions that introduce large overhead. Ignoring the codes with initial overheads below

5%, for which this measure might turn out to be unreliable, LoC 5+ shows varying behavior: In the cases of *104.milc*, *115.fds4*, *130.socorro*, and *Cactus Carpet* very few functions are removed compared to the achieved overhead reduction. For the other applications, the filter is largely ineffective. The cyclomatic filters, by contrast, yield high returns on their removal candidates in the majority of cases. Exceptions are *121.pop2*, *122.tachyon*, and *125.RAxML*. Finally, the effectiveness of MPI Path roughly correlates with the number of functions still instrumented. However, although it removes many functions, it still retains critical information. For example, the detection of MPI call paths that incur waiting time is not affected because all functions on such paths remain instrumented.

6 Conclusion and Future Work

In this paper, we presented an effective approach to reducing the overhead of direct instrumentation for the purpose of parallel performance analysis. Based on structural properties of the program, including both a function's internal structure and/or its external calling relationships, we are able to identify the most significant sources of overhead and remove them from instrumentation. Our solution, which was implemented as a generic and configurable binary instrumenter, requires neither any expensive extra runs nor re-compilation of the target code. We demonstrated that, depending on the analysis objective, in almost all of our test cases the overhead could be reduced to only a few percent. Overall, the MPI Path filter was most effective, allowing low-overhead measurements of the communication behavior across a wide range of applications – except for one malign case with MPI calls in rarely executed error handlers. Moreover, if the focus lies on computation, CC 3+ offers a good balance between the number of excluded functions and the overhead reduction achieved. Finally, the union of MPI Path and CC 3+ seems also promising and should be tried if investigating correlations between the computational load and the communication time is an analysis goal. Whereas this study only considered parallelism via MPI, future work will concentrate on filter rules also suitable for OpenMP applications. A particular challenge to be addressed will be the non-portable representation of OpenMP constructs in the binary.

Acknowledgment. We would like to thank the developer team of the Dyninst library, especially Madhavi Krishnan and Andrew Bernat, for their continuous support.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6), 685–701 (2009)
2. Ball, T., Larus, J.R.: Efficient path profiling. In: Proc. of the 29th ACM/IEEE International Symposium on Microarchitecture. pp. 46–57. IEEE Computer Society, Washington, DC, USA (1996)

3. Buck, B., Hollingsworth, J.: An API for runtime code patching. Journal of High Performance Computing Applications 14(4), 317–329 (2000)
4. Cactus code (2010), http://www.cactuscode.org
5. Gadget 2 (2010), http://www.mpa-garching.mpg.de/gadget
6. Geimer, M., Shende, S.S., Malony, A.D., Wolf, F.: A generic and configurable source-code instrumentation component. In: Proc. of the International Conference on Computational Science (ICCS). LNCS, vol. 5545, pp. 696–705. Springer (May 2009)
7. Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (Apr 2010)
8. Hernandez, O., Jin, H., Chapman, B.: Compiler support for efficient instrumentation. In: Proc. of the ParCo 2007 Conference. Advances in Parallel Computing, vol. 15, pp. 661–668 (2008)
9. JuRoPA (2010), http://www.fz-juelich.de/jsc/juropa
10. Malony, A.D., Shende, S.S.: Overhead compensation in performance profiling. In: Proc. of the 10th International Euro-Par Conference. pp. 119–132. LNCS, Springer (2004)
11. Malony, A.D., Shende, S.S., Morris, A., Wolf, F.: Compensation of measurement overhead in parallel performance profiling. International Journal of High Performance Computing Applications 21(2), 174–194 (2007)
12. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering 2, 308–320 (1976)
13. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: A tool for top-down analysis of node performance. The Journal of Supercomputing 23(1), 81–104 (August 2002)
14. Message Passing Interface Forum: MPI: A message-passing interface standard, version 2.2 (September 2009), chapter 14: Profiling Interface
15. an Mey, D., et al.: Score-P – A unified performance measurement system for petascale applications. In: Proc. of Competence in High Performance Computing, Schloss Schwetzingen, Germany (2010), (to appear)
16. Müller, M., van Waveren, M., Lieberman, R., Whitney, B., Saito, H., Kumaran, K., Baron, J., Brantley, W., Parrott, C., Elken, T., Feng, H., Ponder, C.: SPEC MPI2007 – An application benchmark suite for parallel systems using MPI. Concurrency and Computation: Practice and Experience 22(2), 191–205 (Feb 2010)
17. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
18. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: An open source infrastructure for parallel performance analysis. Scientific Programming 16(2-3), 105–121 (2008)
19. Servat, H., Llort, G., Giménez, J., Labarta, J.: Detailed performance analysis using coarse grain sampling. In: Euro-Par 2009 - Parallel Processing Workshops, LNCS, vol. 6043, pp. 185–198. Springer (2010)
20. Shende, S.S.: The role of instrumentation and mapping in performance measurement. Ph.D. thesis, University of Oregon (August 2001)
21. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
22. Williams, C., Hollingsworth, J.: Interactive binary instrumentation. IEEE Seminar Digests 2004(915), 25–28 (Jan.)