Comparative Evaluation of Call Graph Generation by Profiling Tools

Onur Cankur and Abhinav Bhatele

Department of Computer Science
University of Maryland, College Park, Maryland 20742 USA
ocankur@umd.edu, bhatele@cs.umd.edu

Abstract. Call graphs generated by profiling tools are critical to dissecting the performance of parallel programs. Although many mature and sophisticated profiling tools record call graph data, each tool is different in its runtime overheads, memory consumption, and output data generated. In this work, we perform a comparative evaluation study on the call graph data generation capabilities of several popular profiling tools – Caliper, HPCToolkit, TAU, and Score-P. We evaluate their runtime overheads, memory consumption, and generated call graph data (size and quality). We perform this comparison empirically by executing several proxy applications, AMG, LULESH, and Quicksilver on a parallel cluster. Our results show which tool results in the lowest overheads and produces the most meaningful call graph data under different conditions.

Keywords: profiling tools · call graph · performance analysis · parallel performance · measurement

1 Introduction

Analyzing and optimizing the performance of parallel programs is critical to obtaining high efficiency on high performance computing (HPC) architectures. The complexity in hardware architectures and system software makes measuring and recording performance data challenging. At the same time, the complexity in HPC applications and compiler transformations can make analyzing and attributing performance to source code and external libraries challenging [23]. Even so, a plethora of performance analysis tools exists for gathering and analyzing performance data [5, 1, 9, 22, 13]. One category of performance tools collects performance data that is aggregated over time. In this work, we refer to these as profiling tools to distinguish them from tracing tools that gather more detailed time-series data or full execution traces. Specifically, we focus on profiling tools that record contextual information about the performance data such as calling context, file and line numbers in the source code, etc., which can help users in attributing performance to source code.

Although several profiling tools exist in the HPC community, they differ in their profiling methods and capabilities, which affects their efficiency and the quality of generated performance data. Broadly, profiling tools use one of two

methods for collecting information – instrumentation and sampling. Instrumentation involves adding extra instructions to the source or binary code that are used to measure the execution time of different parts of a program. In contrast, sampling does not require adding instructions. It periodically samples the program counter, uses that to identify the code being executed, and aggregates the performance measurements of a code block across multiple samples. These different profiling methods can lead to varying overheads and capabilities in different profiling tools. For example, an instrumentation-based profiling tool might cause more overhead than a sampling-based tool while providing more accurate output. Besides, two different profiling tools that use the same method might have different capabilities depending on how well they are implemented. For instance, a sampling-based tool might work better than others under low sampling intervals. Since end-users have many choices when using a profiling tool, a systematic study is needed to understand the impact of different profiling techniques on data generation.

Performance data gathered by profiling tools consist of different kinds of information about the program such as the call graph, communication patterns, and MPI process topology. In this study, we focus on the call graph data generation capabilities of profiling tools since the call graph provides critical information about program structure, which can be quite useful in performance analysis. There are many factors that come into play when comparing call graph data generation. Runtime overhead and memory consumption are two comparison metrics that naturally come to mind since they directly impact the application being profiled. In addition, profiling complex parallel applications on a large number of processes can result in a large amount of call graph data being generated, which can also be an important factor to consider when comparing tools. The quality and usefulness of the data generated in terms of its correctness (e.g., correctly measuring and attributing execution time) and ability to attribute performance to source code are also important. In this paper, we consider runtime overhead, memory usage, and quality of the call graph data to compare the data generation capabilities of profiling tools.

We compare several popular tools that are used in the HPC community to profile parallel programs – Caliper [5], HPCToolkit [1], Score-P [9], and TAU [22] – in terms of their capabilities, performance, and generation of meaningful call graph data. More specifically, we compare their runtime overheads, memory usage, and the size, correctness and quality of the generated call graph data. We conduct these experiments on a parallel cluster by profiling three different proxy applications, AMG, LULESH, and Quicksilver, using both instrumentation and sampling under different sampling intervals and different numbers of processes. To the best of our knowledge, this is the first comparative evaluation study on call graph data generation capabilities of profiling tools for parallel programs. In addition, we extend and use Hatchet [4], a Python-based tool that enables analyzing the output from profiling tools programmatically, to compare call graph data. We show which tools are more efficient in terms of measurement overheads and memory consumption, and generate more meaningful call graph data under

different conditions and for different proxy applications. Specifically, this study makes the following contributions:

- Comparatively evaluate the call graph generation capabilities of profiling tools considering their measurement and memory overheads, and quality of the generated data.
- Extend the Hatchet performance analysis tool to support output data generated by Score-P and TAU, enabling the comparison of data from several popular profiling tools.
- Provide feedback to tool developers for the improvement of various aspects of the performance data gathering process.

2 Background and Related Work

In this section, we provide an overview of the profiling tools used in this paper and give detailed background information about profiling methods and the output of profiling tools. We also introduce Hatchet, using which we perform our analyses. Finally, we present related work on the evaluation of profiling tools.

2.1 Different Methods for Profiling

Performance measurement tools can be divided into two categories: profiling and tracing. In this work, we only consider profiling which can be done using sampling or instrumentation. Instrumentation can be classified along two dimensions: the method of instrumentation and where is the instrumentation added. The method can be manual (by the developer) or automatic (by the tool, compiler, library interposition, etc.) and it can be performed by adding additional instructions in the source code, byte code, or binary code [18]. These additional instructions allow measuring the performance of a code section.

On the other hand, sampling-based profiling tools take periodic snapshots of the program, check the location of the program counter and collect the function call stack by performing stack unwinding [23] and then aggregate this data that they gathered periodically. It also allows to change sampling interval, hence, provides controllable overhead.

2.2 Information Gathered by Profiling Tools

The data generated by profiling tools usually contains contextual information (the function name, file name, call path, process or thread ID, etc.) and performance metrics such as time, cache misses, communication information, and the total number of instructions along with the callpath information on the program. Some profiling tools collect individual callpaths (i.e. calling contexts) on the program and represent it in a tree format called calling context tree. Other tools aggregate that information and generate call graphs which show aggregated information in which a procedure that is called in multiple distinct paths

is represented as a single node. Profiling tools typically have their own custom output formats to store the calling context tree (CCT) or call graph. In this paper, we use call graph as a general term for both CCT and call graph.

2.3 Profiling Tools Used in this Study

All profiling tools in this study, which are introduced below, support C, C++, and Fortran programs and MPI, OpenMP, and pthreads programming models (see Table 1).

Table 1: Salient features of different profiling tools

Tool	Samp.	Instr.	Languages	Output Format
Caliper	Yes	Yes	C, C++, Fortran	.json and custom
HPCToolkit	Yes	No	C, C++, Fortran	XML and custom
Score-P	Partially	Yes	C, C++, Fortran, Python	XML and custom
TAU	Yes	Yes	C, C++, Fortran, Java, Python	custom

Caliper is a performance analysis toolbox that provides many services for users to measure and analyze the performance, such as tracing and profiling services [5]. It allows users to activate these capabilities at runtime by annotating the source code or using configuration files. It provides *json* or custom file formats and generates CCT data.

HPCToolkit is a toolkit for performance measurement and analysis [1]. It supports both profiling and tracing and uses sampling instead of instrumentation. It generates a performance database directory that contains *XML* and custom file formats that store CCT information.

Score-P is a measurement infrastructure that supports both profiling and tracing [9]. It is primarily an instrumentation-based tool that supports source, compiler, and selective instrumentation, but it also supports sampling for instrumented executables. Score-P supports Python in addition to C, C++, and Fortran. It generates `.cubex` [20] output tarballs which are in *CUBE4* format and contain files in *XML* and custom formats and generates CCT information.

TAU is also a performance measurement and analysis toolkit and supports profiling and tracing [22]. TAU also primarily uses instrumentation but it also supports sampling. It supports different types of instrumentation such as source instrumentation using PDT [11], compiler instrumentation, and selective instrumentation. It supports C, C++, Fortran, Java, and Python and generates `profile.<rank>.<>.<thread>` files which are in custom format and stores CCT as its default profiling output format.

2.4 Post-mortem Analysis of Profiling Data

Most of the profiling tools we evaluate provide their own analysis and visualization tools such as HPCViewer [15], ParaProf [3], and CubeGUI [20]. Visualization

tools usually provide a graphical user interface (GUI) that allows the visualization of one or two call graphs at the same time. These GUIs provide limited call graph analysis capabilities since they do not provide a programmable interface. In this study, we used and improved Hatchet [4] to compare the call graph data generated from different profiling tools on the same platform.

Hatchet is a Python-based performance analysis tool that provides a programmable interface to analyze the call graph profiling data of different tools on the same platform [4]. It reads in the profiling data and generates a data structure called *graphframe* which stores numerical (e.g. time, cache misses) and categorical (callpath, file and line information, etc.) information along with the caller-callee relationships on the program.

2.5 Related Work

All tools used in this paper have some kind of prior performance evaluation. For example, some of them study the overhead of TAU using tracing, profiling, sampling, and instrumentation [21, 17, 14]. There is a similar study on HPCToolkit [12] which includes runtime overhead evaluation. Score-P and Caliper include similar runtime overhead evaluation studies in their corresponding papers [9] [5]. Although each tool has been evaluated for performance, these past studies only cover the runtime and memory overhead of a tool, different profiling methods a tool supports, or include a simple overhead comparison with another tool that is not currently state-of-the-art. Therefore, the only criteria considered in these papers are the runtime and memory overheads, and they do not evaluate the quality of the call graph data generated by profiling tools.

Other evaluation studies on profiling tools only include functional comparisons [8, 16]. The closest related work to our paper is published in 2008 [10]. However, it is more like a case study and a generic user experience comparison of profiling tools that were widely used at that time and it does not contain empirical experiments. Our study is the first empirical comparative study on call graph data generation by state-of-the-art profiling tools, considering their runtime overhead, memory usage, and output quality.

3 Methodology for Comparative Evaluation

In this study, we consider runtime overhead, memory usage, size, richness and correctness of the generated call graph data. We do not consider information such as communication volume and process topology. Below, we present the various axes along which call graph data generation capabilities are compared and describe the metrics used for comparison.

3.1 Comparison of Runtime Overhead

One of the most important factors to consider when comparing call graph data generation is the runtime overhead incurred when using them. The execution

time of an application should not be perturbed significantly by the profiling tool. Different profiling methods can incur different overheads. For example, sampling causes less overhead than instrumentation methods because it is less intrusive. Similarly, one instrumentation method can cause more overhead than another instrumentation method. Hence, we evaluate the runtime overhead by conducting experiments using both sampling and instrumentation techniques separately. In addition, sampling-based methods have the flexibility to adjust the runtime overhead by increasing or decreasing the sampling interval. We also compare the tools by varying the sampling intervals wherever supported.

We run each application without any profiling and measure the execution time by calling `MPI_Wtime()` at the start and end of the program. Dividing these two timings gives us the relative execution time of a program. We then run each application with different profiling tools to calculate the increase in execution time due to profiling overheads.

3.2 Comparison of Memory Consumption

Ideally, performance tools should not consume large amounts of memory. Hence, it is important to compare the additional memory consumption of different profiling tools. We compare the amount of memory consumed by profiling tools during application execution. We perform the same experiments using the default and varying sampling intervals and using instrumentation.

We measure the memory usage of a program using the `getrusage()` function call and obtain the largest memory usage at any point during program execution. We calculate the additional memory consumed by a tool by gathering memory usage information for two runs – one with and one without profiling.

3.3 Comparison of the Quality of Call Graph Data

We expect profiling tools to provide useful information without generating unnecessary or repetitive information. In this study, we evaluate the quality of the call graph profiling data recorded by each tool considering the data size, correctness and richness of the data with the assumption that if the data generated by multiple tools is nearly identical, it should be close to the ground truth.

Size of Call Graph Data: A significant amount of call graph data can be generated when profiling HPC applications, which can make post-mortem analysis challenging. We evaluate the size of the data generated when using different tools for the same experiments by using default and varying sampling intervals and instrumentation. We use default settings for each tool without changing the number of metrics collected and collect per-process data without aggregating it. We also observe how the data size increases with an increase in the number of processes since some tools generate a separate file per MPI process while others represent this data in a more compact output format.

Correctness of Call Graph Data: The correctness of the generated call graph data is critical in order to perform meaningful analysis. We consider the infor-

mation to be correct if different tools generate the same results with correct contextual information. We follow two different strategies for this analysis. First, we load the profiling data from different tools in Hatchet and identify the top 5 slowest call graph nodes in the call graph by inclusive and exclusive time and investigate if the tools identify the same slowest nodes. We also compare the file, line numbers, and callpaths reported by each tool for the slowest nodes. Second, we identify the hot path in each dataset. The hot path refers to a call path in the graph in which all nodes account for 50% or more of the inclusive time of their parent [2]. The node at the end of a hot path is called a hot node. Therefore, hot path analysis gives us the most time-consuming call path in the execution. Our hot path analysis implementation in Hatchet makes it possible to perform the same analysis for each tool.

Richness of Call Graph Data: The richness of call graph profiling data refers to having detailed information in the CCT such as caller-callee relationships, and contextual information (file and module information, line number, etc.). To evaluate richness, we take the following parameters into account: the maximum and average callpath lengths, the number of nodes, the number of identified .so files (dynamically loaded modules), and the number of MPI routines. The callpath length provides insight into how detailed the caller-callee relationships are. In addition, examining the number of total and unique nodes in the call graph tells us if a tool is missing some information or generating excessive data that is not required. We also compare the information generated by different tools about dynamically loaded libraries and MPI routines. Similar to the correctness evaluation, these comparisons are performed using Hatchet. For example, we filter the Hatchet dataframe by node names to get the MPI functions or .so files. We traverse the graph to calculate the maximum and average callpath length.

3.4 Extensions to Hatchet

We have improved Hatchet by implementing TAU and Score-P readers to use in this study. Below, we explain how we implement these readers.

Score-P Reader: Score-P stores profiling data in CUBE4 tar files (extension: `.cubex`) [20]. These tar files in turn contain `anchor.xml`, `.index`, and `.data` files. The `anchor.xml` file contains metadata information about metrics and processes along with caller-callee relationships. The `.index` and `.data` files contain information about metric measurement and metric values. To implement a Python reader in Hatchet, we use pyCubexR which is a Score-P reader that can read `cubex` files. After implementing the reader, we compared the generated Hatchet graphframe with the CubeGUI visualization provided by Score-P to confirm the correctness of our implementation.

TAU Reader: TAU generates profiles in its custom format. It generates a separate file for each process and thread. In addition, it generates a separate directory for each metric (time, cache misses, etc.). We combine all this information gathered from different directories and files, and create a single CCT which is stored

as a graphframe in Hatchet. Finally, we validate our reader implementation by comparing the Hatchet graphframe with ParaProf output which is a visualization tool for TAU outputs.

4 Experimental Setup

In this section, we describe each experiment in detail. We used three HPC applications written in C/C++ and four profiling tools in our experiments: AMG [6], LULESH [7], and Quicksilver [19] proxy applications and Caliper, HPCToolkit, Score-P, and TAU profiling tools. We chose LULESH because it is a simple code (lines of code=∼5.5k), which can help us illustrate differences between tools. Quicksilver is more complex than LULESH in terms of lines of code (∼10k), and AMG (∼65k) uses external libraries such as Hypre which makes its call paths more complex. In addition, all the profiling tools we used in this study are actively and widely used in many supercomputers and they are still being improved. We used the latest release versions of these tools: Caliper 2.6.0, HPCToolkit 2021.05.15, Score-P 7.1, and TAU 2.30.1. We compared their sampling and instrumentation capabilities by running experiments accordingly. We separately built each of the applications with these tools using GCC 8.3.1 and Open MPI 3.0.1. We only used MPI, so multithreading using OpenMP or Pthreads was not enabled. We ran the applications on a parallel cluster which has x86_64 architecture with 36 cores on each node and performed weak scaling experiments using 64, 128 (125 for LULESH), 256 (216 for LULESH), and 512 processes using 1 through 16 nodes and 32 cores on each node in all experiments.

4.1 Experiment 1: Comparison of Sampling Capabilities

In this experiment, we used Caliper, HPCToolkit, Score-P, and TAU using their default sampling intervals. However, it should be noted that Score-P supports sampling of instrumented programs, while other tools directly perform sampling on executables without instrumenting them. The default sampling interval for Caliper, HPCToolkit, Score-P, and TAU is 20, 5, 10, and 30 ms, respectively. We evaluated the runtime overhead, memory usage and the size, richness, and correctness of the generated data.

4.2 Experiment 2: Impact of Sampling Intervals

Similar to Experiment 1, we only used the tools that support sampling and evaluated the same comparison metrics. However, for this experiment, we used varying sampling intervals as follows: 1.25, 2.5, 5, 10, 20 ms. Sampling interval refers to the milliseconds spent between two samples (Caliper uses Hertz as a unit). For example, sampling interval with a value of 5 ms refers that the profiling tool samples the program for every 5 milliseconds. This experiment shows whether tools can properly work when the sampling interval is low and how the performance and the generated data change as we change the interval.

4.3 Experiment 3: Comparison of Instrumentation Capabilities

In this experiment, we compared the instrumentation capabilities of Caliper, Score-P, and TAU since HPCToolkit does not support instrumentation. We tried to use the default instrumentation method that the tools support. By default, Caliper supports manual source instrumentation, Score-P supports compiler instrumentation, and TAU supports automatic source instrumentation. During the experiments, we realized that TAU's automatic source instrumentation, which uses PDT, gives errors for almost all of the runs because it is not fully updated. Therefore, we decided to use compiler instrumentation for TAU which works for all applications. Caliper requires manual annotations to perform the source instrumentation. We used annotated versions of LULESH and Quicksilver which are publicly shared by Caliper developers on Github and we annotated AMG by ourselves learning from the already available annotations. We evaluated the same comparison metrics also in this experiment and this experiment shows which tool or instrumentation method causes more overhead or can generate better data and how well these tools can perform an instrumentation method.

5 Evaluation

In this section, we present the findings of our empirical comparison of call graph data generation by different profiling tools.

5.1 Runtime Overhead

We first evaluate the runtime overhead of the profiling tools by performing experiments using instrumentation and sampling with default and varying intervals.

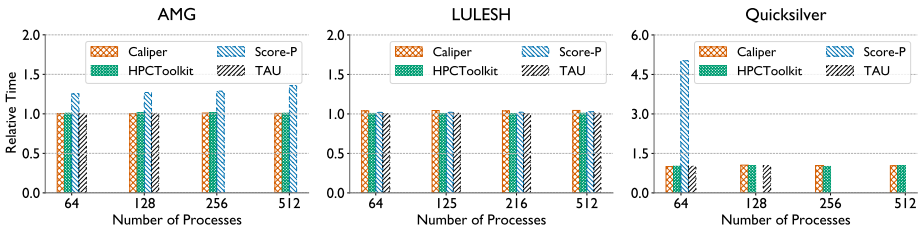


Fig. 1: Runtime overhead for different tools when the sampling method is used. Default sampling intervals (20, 5, 10, and 30 ms) were used for Caliper, HPCToolkit, Score-P, and TAU respectively.

Figure 1 shows runtime overheads caused by Caliper, HPCToolkit, Score-P, and TAU when we sample the programs using default sampling intervals. We can see that most tools have a small overhead (slightly over 1x) except Score-P. Score-P has ~1.25x overhead for AMG, ~1.02x for LULESH, and ~5x for Quicksilver.

We think that the significant difference between the overhead caused by Score-P and other tools is because Score-P samples instrumented executables while others can directly perform sampling on uninstrumented executables. Caliper, HPCToolkit, and TAU do not have a significant overhead. The overhead increases as we increase the number of processes but the increase is small. In addition, TAU fails to produce output when we use it with AMG and Quicksilver on 256 and 512 processes. Similarly, Score-P does not work when we run Quicksilver using 128, 256, and 512 processes. Both TAU and Score-P give segmentation faults in some runs. We tried to fix these errors by debugging, running the applications multiple times, and contacting the developers but could not find a solution.

In Figure 2, we show the runtime overhead of the tools under varying sampling intervals (1.25–20.0 ms). It can be observed that the runtime overhead does not change significantly under different sampling intervals and the results are similar to what we see in Figure 1. Hence, we can say that the sampling interval does not have a significant impact on the runtime overhead. We realized that Caliper, Score-P, and TAU do not work at all when the sampling interval is 1.25 ms on AMG and Quicksilver runs, and TAU and Score-P do not work deterministically under some sampling intervals. For example, sometimes three of five experiments run to completion while at other times, only one of them works. HPCToolkit works under all samplings intervals and its runtime overhead is stable in all settings.

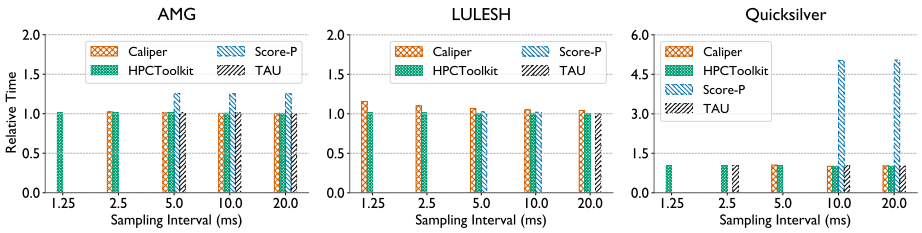


Fig. 2: Runtime overhead for different tools as a function of the sampling interval used. Each execution used 64 MPI processes.

Figure 3 shows the runtime overhead caused by Caliper, Score-P, and TAU when the instrumentation method is used instead of sampling. As mentioned before, we used automatic compiler instrumentation for both TAU and Score-P and manual source instrumentation for Caliper. All three plots in Figure 3 show that Caliper results in lower runtime overhead. Interestingly, TAU has the highest runtime overhead (~2x) for LULESH while Score-P has the highest overhead for Quicksilver (~10x) although the same compiler version and same compiler wrappers that the tools provide are used. We believe this is related to the implementation details of each tool and how they handle some specific cases (e.g. inlining and loop optimizations). Therefore, we can say that compiler instrumentation is not stable under different conditions and is highly dependent on

the application. We also note that TAU and Score-P's compiler instrumentation of Quicksilver causes more overhead compared to sampling (Figure 1).

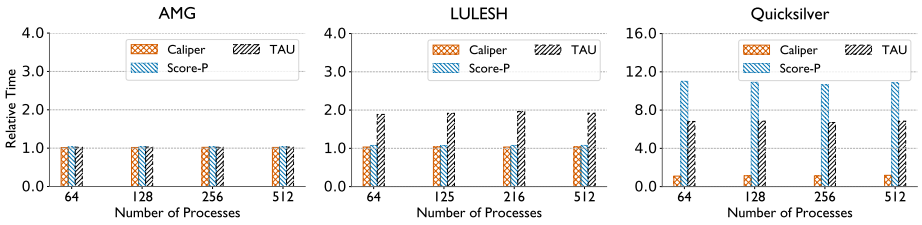


Fig. 3: Runtime overhead for different tools when the instrumentation method is used. Caliper uses source instrumentation, while Score-P and TAU use compiler instrumentation.

5.2 Memory Consumption

In this section, we evaluate the memory usage of each tool but we do not report the results under default sampling intervals because we observed that it does not change significantly depending on the sampling interval.

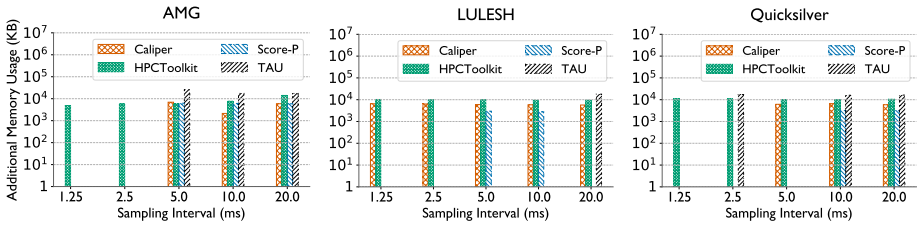


Fig. 4: Total additional memory usage (in KB) for different tools as a function of the sampling interval used. Each execution used 64 MPI processes.

Figures 4 and 5 show that the total memory usage for each tool typically does not change drastically with different applications, different numbers of processes, and different sampling intervals. It can be observed in both figures that TAU uses more memory in all of the runs where it works (~10 MB in sampling and ~100 MB in instrumentation) compared to the other tools. Score-P has the least memory usage except in AMG runs using 10 ms and 20 ms sampling intervals (see left plot in Figure 4). HPCToolkit has the second-highest memory usage while Caliper has the third-highest. It can also be seen that TAU uses more memory when compiler instrumentation is used (~100 MB, Figure 5) versus sampling (~10 MB, Figure 4). Because of this significant difference between tools, we can say that memory usage is an important comparison metric to evaluate call graph data generation.

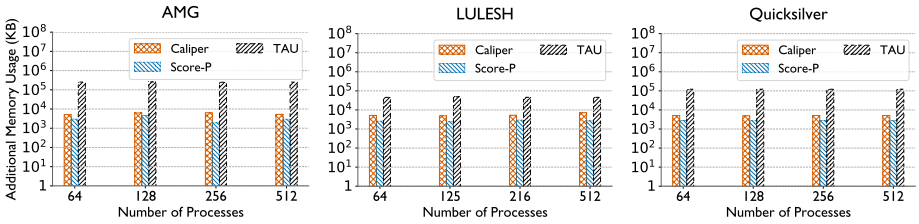


Fig. 5: Total additional memory usage (in KB) for different tools when the instrumentation method is used. Caliper uses source instrumentation, while Score-P and TAU use compiler instrumentation.

Next, we evaluate the quality of the generated call graph data considering the size, richness, and meaningfulness of the data.

5.3 Size of Call Graph Data

We compared the size of the generated call graph data while performing the same experiments. We observed that there is a significant difference between tools in terms of the size of the generated data.

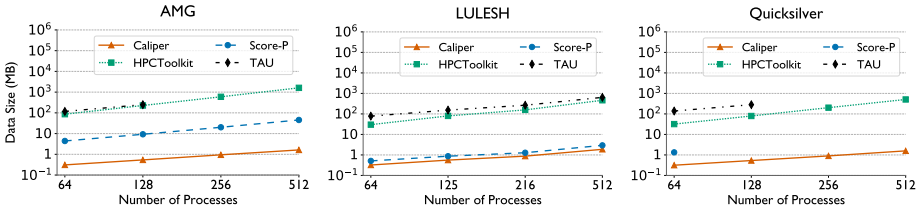


Fig. 6: Size of the profiling data (in MB) for different tools when the default sampling method is used. Default sampling intervals (20, 5, 10, and 30 ms) were used for Caliper, HPCToolkit, Score-P, and TAU respectively.

Figure 6 shows the increase in the data size when the default sampling method is used for each tool. The size of the generated data increases with an increase in the number of processes since data for more processes is being recorded. We can see that TAU generates the largest amount of data for all applications (from ~100 to ~1000 MB). In addition, TAU and HPCToolkit generate much more data compared to Score-P and Caliper because they generate a separate file for each process while Score-P and Caliper generate more compact data. For example, Caliper generates only a single *json* file that contains information about all the processes. In contrast, Figure 7 shows the decrease in the data size under varying sampling intervals. In this case, the size of the data decreases as we increase the sampling interval since less data is being recorded as we increase the time between two samples. Interestingly, Caliper has an opposite behavior and it generates slightly more data as the sampling interval is

increased. We examined the Caliper data and realized that it generates more nodes as we increase the interval up to 5.0 ms and then, it starts generating fewer nodes again.

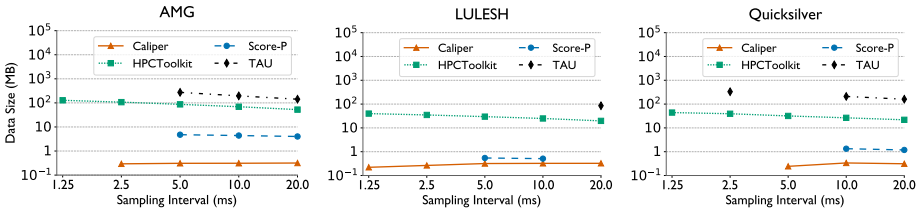


Fig. 7: Size of the profiling data (in MB) for different tools as a function of the sampling interval used. Each execution used 64 MPI processes.

Similar to Figure 6, we see the increase in data size when using instrumentation in Figure 8. As the number of processes is increased, TAU generates the largest amount of data. In addition, it can be also seen from LULESH plots in Figures 6 and 8 that TAU generates more data when sampling is used instead of compiler instrumentation because it generates additional information such as `[CONTEXT] and [SAMPLE]` nodes. `[CONTEXT]` nodes do not store useful information and they can be removed from the data.

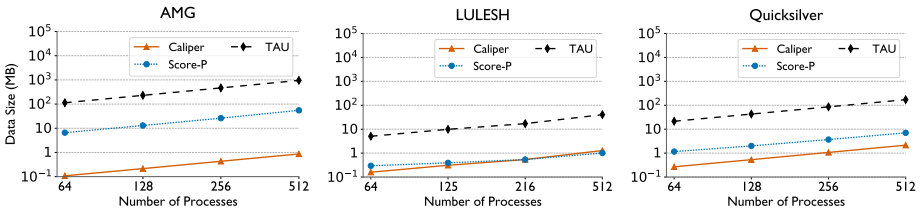


Fig. 8: Size of the profiling data (in MB) for different tools when the instrumentation method is used. Caliper uses source instrumentation, while Score-P and TAU use compiler instrumentation.

5.4 Correctness of Call Graph Data

In order to evaluate the correctness of the call graph data, we compare the two slowest nodes and their identified callpaths and summarize the other findings. We only report results for LULESH since we get similar results with other applications. First, we identified the slowest and the hot node for each tool and checked if the file and line information of the slowest node is correct. Table 2 shows the identified slowest nodes, hot nodes, and the correctness of file and line information for the slowest node. We assume that if the majority of the tools provide the same output, it should be close to the ground truth. As it can be seen from the table, Caliper instrumentation, HPCToolkit sampling, and TAU

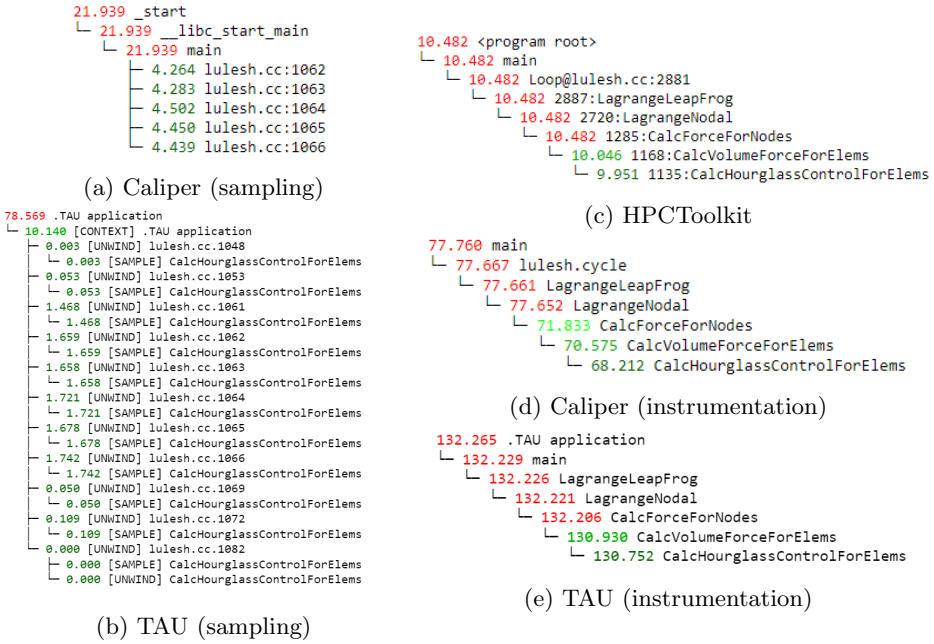
instrumentation can identify the same node, CalcHourglassControlForElems, as the slowest node with the correct file and line information. Caliper and TAU sampling cannot identify the same node as the slowest node although they also have the same node in their output data with the correct file and line information which suggests that either these tools record a different time value for that node or they have incomplete contextual information. The CalcHourglassControlForElems node was missing in Score-P output, therefore, we could not identify it. We also observed that Caliper instrumentation cannot generate file and line information but we could not check that for Score-P since the node was missing in its output. Score-P does not identify the same slowest node because it does not record information for inlined functions by default but provides an option to enable it.

Table 2: Comparison of the correctness of the generated call graph data for different tools when the default sampling interval and instrumentation method are used. The data was generated by executing LULESH using 64 MPI processes.

Tool	Method	Slowest node (inc. time, exc. time)	Hot node	File & line correctness
Caliper	Sampling	(main, syscall)***	main***	Correct**
	Instrumentation	(main, CalcHourglassControlForElems)	lulesh.cycle	Missing
HPCToolkit	Sampling	(main, CalcHourglassControlForElems)	Loop in lulesh.cc at line 1048. (CalcHourglassControlForElems)	Correct Correct
Score-P	Sampling	(lulesh-scorep2.0 (main), main)	lulesh-scorep2.0 (main)	Missing
	Instrumentation	(lulesh-scorep2.0 (main), main)	main	Missing
TAU	Sampling	(progress_engine, progress_engine)***	.TAU Application***	Correct**
	Instrumentation	(.TAU Application (main), CalcHourglassControlForElems)	CalcHourglassControlForElems	Correct

Figure 9 shows the callpath for the commonly identified slowest node, CalcHourglassControlForElems. We confirm that TAU and Caliper sampling outputs (Figure 9(b), 9(a)) contain information about that node and can generate its callpath although they cannot identify it as the slowest node. We can see from TAU and Caliper sampling callpaths that they do not aggregate the measured time values for that node and they connect the nodes related to it directly to the main node which results in having many related nodes with low time values. In addition, Caliper sampling cannot generate the name of the node as seen in Figure 9(a), and the only way to find it is to use the line information on the output for that node. Score-P is missing that node in its output, therefore, it is not included in this figure. In summary, TAU and Caliper sampling and Score-P generate incomplete call graphs for LULESH compared to TAU instrumentation, Caliper instrumentation, and HPCToolkit.

We also investigated the top five slowest nodes and observed that Caliper instrumentation, HPCToolkit, and TAU instrumentation identify almost the same nodes as the top five but the order of the top five list is somewhat different in each tool. Score-P sampling and instrumentation also find similar top five slowest nodes with greater differences. Caliper and TAU sampling do not identify



(a) Caliper (sampling)

(b) TAU (sampling)

(c) HPCToolkit

(d) Caliper (instrumentation)

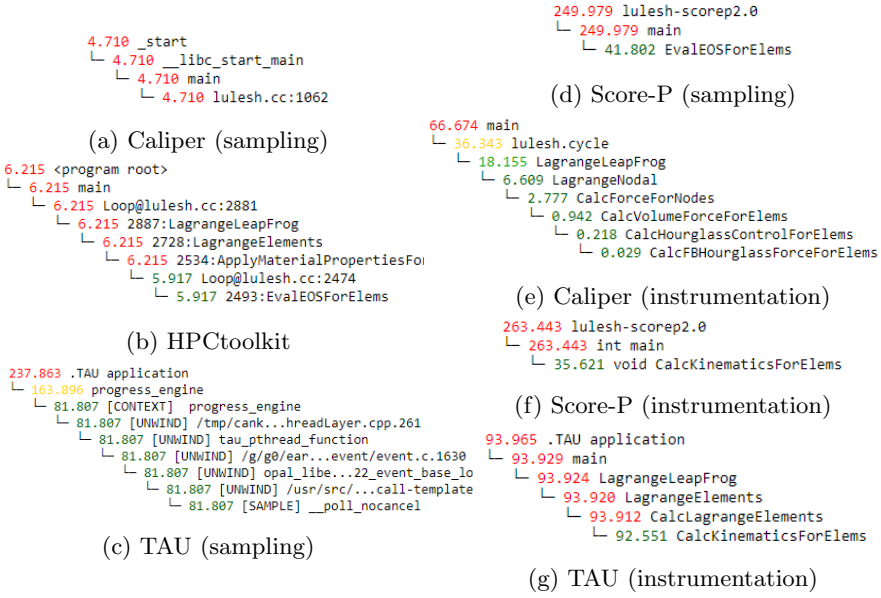
(e) TAU (instrumentation)

Fig. 9: Callpath of the CalcHourglassControlForElems node obtained by different tools for LULESH running on 64 processes.

the same slowest nodes. We present the call paths of the second slowest node in Figure 10. The leaf node in each call path is the second slowest node. It can be seen that the leaf nodes are usually different from each other except for a few similarities. Score-P instrumentation (Figure 10(f)) identifies the same second slowest node as TAU instrumentation (Figure 10(g)). However, their callpaths are not identical since they handle inlined functions differently. Similarly, Score-P sampling (Figure 10(d)) identifies the same second slowest node as HPCToolkit (Figure 10(b)) but the callpaths are different. In addition, Caliper instrumentation (Figure 10(e)) does not identify the same slowest node but the node identified by Caliper is also in the top five list of HPCToolkit and TAU instrumentation. Caliper sampling (Figure 10(a)) and TAU sampling (Figure 10(c)) do not provide as meaningful results. Note that we do not have Score-P results in Figure 9 because Score-P identifies the main node as the slowest node and the slowest node that is commonly identified by other tools does not exist in the Score-P output. However, it identifies the same second slowest node as some other profiling tools, hence, we included Score-P in Figure 10.

5.5 Richness of Call Graph Data

We compared the richness of the call graph data generated by the profiling tools considering the maximum and average callpath depth, the number of nodes,



(a) Caliper (sampling)

(b) HPCtoolkit

(c) TAU (sampling)

(d) Score-P (sampling)

(e) Caliper (instrumentation)

(f) Score-P (instrumentation)

(g) TAU (instrumentation)

Fig. 10: Callpath of the second slowest node obtained by different tools for LULESH running on 64 processes.

the number of dynamically loaded libraries (.so files), and the number of MPI functions. The data is gathered by running each application on 64 processes.

Table 3 shows the richness of the data generated by each tool using their default method. The fourth and fifth columns show the maximum and average callpath lengths in the call graph data. The callpaths generated by HPCToolkit and Caliper sampling usually have similar depths. TAU and Score-P compiler instrumentation abnormally generate very long callpaths for a node called `hypre_qsort0` which is a recursive sorting function. Interestingly, they keep creating a new callpath for that function every time it calls itself instead of aggregating its information. The other tools usually generate callpaths that have fewer than ten nodes. The length of the callpaths might be related to how well a tool can handle inlined functions, but generating unnecessary data might also result in longer callpaths. Therefore, we cannot infer that a longer callpath is richer. In addition, some of these tools allow the user to set the maximum callpath length to be recorded, so expert users could adjust it depending on their needs. Therefore, this comparison gives insights on tools' and profiling methods' capabilities for generating sufficient call graph data with enough caller-callee relationships.

The next column shows the number of all and unique nodes. HPCToolkit data usually contains more unique nodes although TAU sampling usually has the largest number of nodes. We believe that it is related to how `[UNWIND]` nodes are stored in TAU data format since we realized that they include unnecessary information (confirmed by TAU developers). This suggests that the information

Table 3: Comparison of the richness of the generated data for different tools when a fixed sampling interval (20.0 ms) and the instrumentation method are used. Each execution used 64 MPI processes.

App.	Tool	Method	Max depth	Avg depth	No. of nodes (all,unq)	No. of .so files (all,unq)	No. of MPI functions (all,unq)
AMG	Caliper	Sampling	30	9.724	(1414,739)	(363,**36**)	(37,17)
		Instrumentation	3	2.384	(50,22)	0	(38,10)
	HPCToolkit	Sampling	**35**	**13.931**	(12112,**2528**)	(**4616**,25)	(585,66)
	Score-P	Sampling	63	10.859	(1470,199)	0	(668,52)
		Instrumentation	163*	31.428*	(3117,332)	0	(676,51)
	TAU	Sampling	12	8.416	(**13645**,1976)	(2010,20)	(**1036,91**)
		Instrumentation	111*	10.12*	(1956,334)	0	(683,52)
LULESH	Caliper	Sampling	19	3.984	(832,729)	(96,**47**)	(7,6)
		Instrumentation	7	5.115	(71,31)	0	(40,7)
	HPCToolkit	Sampling	**23**	**10.412**	(4546,**1775**)	(**1496**,22)	(96,**81**)
	Score-P	Sampling	5	3.0	(97,65)	0	(19,11)
		Instrumentation	4	2.656	(43,34)	0	(19,11)
	TAU	Sampling	12	5.473	(**4999**,1281)	(915,12)	(**236**,32)
		Instrumentation	8	4.408	(114,78)	0	(36,11)
Quicksilver	Caliper	Sampling	**30**	10.703	(1495,807)	(413,**25**)	(17,8)
		Instrumentation	8	3.937	(122,84)	0	(36,7)
	HPCToolkit	Sampling	29	**14.376**	(5253,**2392**)	(**1307**,22)	(24,15)
	Score-P	Sampling	10	5.05	(343,206)	0	(40,15)
		Instrumentation	9	5.184	(418,267)	0	(80,29)
	TAU	Sampling	12	7.802	(**7776**,1779)	(731,16)	(**230,41**)
		Instrumentation	9	4.831	(401,246)	0	(47,18)

is not stored as efficiently in TAU. Caliper and Score-P usually generate call graphs with fewer nodes since they generate less data.

The difference between the number of all `.so` files generated by different tools is larger than the difference between the number of unique `.so` files. For example, while HPCToolkit output contains a much larger number of `.so` files compared to Caliper sampling, the number of unique `.so` files in Caliper sampling is larger. The reason is that HPCToolkit can identify more dynamically loaded libraries while Caliper can identify only some of them so the number of all `.so` files is much higher in HPCToolkit data. We also realized that the number of unique `.so` files are is significantly different from each other when sampling is used. However, Score-P does not provide information about `.so` files when we use sampling. The table also shows that `.so` files cannot be identified when using instrumentation which is expected since they are dynamically loaded libraries. We emphasize that it does not imply that the instrumentation method provides poor call graph data compared to the sampling method since information about `.so` files might not be necessary for some analyses.

The last column shows the number of MPI functions. We investigated how many MPI functions can be detected by each tool since it is a commonly used programming model. TAU sampling generates a significantly large number of MPI functions in all applications compared to other tools. As mentioned before, the reason might be that TAU generates unnecessary `[CONTEXT]` nodes that do not contain useful information and these nodes are mostly related to MPI functions.

In summary, all the information about runtime overhead, memory usage, and data size should be connected to the quality of output to have a more complete evaluation of call graph data generation. We emphasize that we cannot conclude that a tool provides richer call graph data by only looking at Table 3. However, this comparison shows some characteristics, abnormalities, and sufficiency of call graph data generated by different tools.

6 Discussion

Our comparative evaluation shows that the runtime overhead when using profiling tools is similar, except in the case of Score-P for some applications. Additional memory consumed by a tool does not vary significantly with the application being profiled. In general, we can order the memory usage of tools from highest to lowest as TAU, HPCToolkit, Caliper, and Score-P. Also, TAU typically generates the largest amount of data with HPCToolkit being a close second. The size of the data generated by Score-P and Caliper is notably lower compared to TAU and HPCToolkit because their representation of output data is more compact. The top five slowest nodes identified by the profiling tools are usually similar to each other except when using sampling in Caliper and TAU. However, although different tools identify the same nodes as slow, the relative ordering of the top five slowest nodes is usually different from each other. In terms of call path completeness, Caliper instrumentation, TAU instrumentation and HPCToolkit generate more complete call graphs in default mode.

After extensively using and evaluating the tools, we are also in a position to provide some feedback to their respective developers. From all the figures in Section 5.3, it can be seen that TAU usually generates the largest amount of data. The reason for this is that it stores repetitive information such as `[CONTEXT]` nodes. These nodes do not have useful metric values and could be removed from the generated data. In addition, TAU stores the same metric information twice – in a separate line by itself and at the end of each callpath. This can be optimized by storing the information only once. When we implemented a reader for TAU output in Hatchet, we realized that TAU generates a separate file for each metric that contains exactly the same callpath information when more than one metric is measured. The size of the output data can be further reduced by storing the call graph only once.

When using the instrumentation method in Caliper, we observed that that Caliper does not generate file and line number information in instrumentation only mode. Although we perform manual source instrumentation in this study, it would be helpful for the end user if file and line number information was in the output. Finally, when using sampling in Caliper and TAU and either method in Score-P, the generated call graphs are relatively incomplete on the experiments performed in this study. We believe that their callpath generation capabilities can be improved.

In summary, we performed the first empirical study to compare call graph data generation capabilities of profiling tools considering many different aspects.

We used these tools as per their official documentation and contacted the tool developers when needed. This study shows that more comprehensive evaluation studies on profiling tools considering their scalability and other performance analysis capabilities may reveal interesting information and could be helpful for the community. In the future, we plan to extend this work by using production applications, collecting other structural information, and performing more empirical and analytical analyses on the output data.

Acknowledgments

This work was supported by funding provided by the University of Maryland College Park Foundation.

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 (2010)
2. Adhianto, L., Mellor-Crummey, J., Tallent, N.R.: Effectively presenting call path profiles of application performance. In: 2010 39th International Conference on Parallel Processing Workshops. pp. 179–188. IEEE (2010)
3. Bell, R., Malony, A.D., Shende, S.: Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In: European Conference on Parallel Processing. pp. 17–26. Springer (2003)
4. Bhatele, A., Brink, S., Gamblin, T.: Hatchet: Pruning the overgrowth in parallel profiles. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19 (Nov 2019), http://doi.acm.org/10.1145/3295500.3356219, lLNL-CONF-772402
5. Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.T., Gimenez, A., LeGendre, M., Pearce, O., Schulz, M.: Caliper: Performance introspection for hpc software stacks. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 550–560 (2016). https://doi.org/10.1109/SC.2016.46
6. Henson, V.E., Yang, U.M.: Boomeramg: A parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics **41**(1), 155–177 (2002). https://doi.org/https://doi.org/10.1016/S0168-9274(01)00115-5, https://www.sciencedirect.com/science/article/pii/S0168927401001155, developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss
7. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)
8. Knobloch, M., Mohr, B.: Tools for gpu computing–debugging and performance analysis of heterogenous hpc applications. Supercomputing Frontiers and Innovations **7**(1), 91–111 (2020)

9. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011. pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
10. Leko, A., Sherburne, H., Su, H., Golden, B., George, A.D.: Practical experiences with modern parallel performance analysis tools: an evaluation. In: Parallel and Distributed Processing, IPDPS 2008 IEEE Symposium. pp. 14–18 (2008)
11. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A tool framework for static and dynamic analysis of object-oriented software with templates. In: SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. pp. 49–49. IEEE (2000)
12. Liu, X., Mellor-Crummey, J.: A tool to analyze the performance of multithreaded programs on numa architectures. ACM Sigplan Notices **49**(8), 259–272 (2014)
13. Madsen, J.R., Awan, M.G., Brunie, H., Deslippe, J., Gayatri, R., Oliker, L., Wang, Y., Yang, C., Williams, S.: Timemory: modular performance analysis for hpc. In: International Conference on High Performance Computing. pp. 434–452. Springer (2020)
14. Malony, A.D., Huck, K.A.: General hybrid parallel profiling. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 204–212. IEEE (2014)
15. Mellor-Crummey, J., Fowler, R., Marin, G.: HPCView: A tool for top-down analysis of node performance. The Journal of Supercomputing **23**, 81–101 (2002)
16. Mohr, B.: Scalable parallel performance measurement and analysis tools-state-of-the-art and future challenges. Supercomputing frontiers and innovations **1**(2), 108–123 (2014)
17. Nataraj, A., Sottile, M., Morris, A., Malony, A.D., Shende, S.: Tauoversupermon: Low-overhead online parallel performance monitoring. In: European Conference on Parallel Processing. pp. 85–96. Springer (2007)
18. Nethercote, N.: Dynamic binary analysis and instrumentation. Tech. rep., University of Cambridge, Computer Laboratory (2004)
19. Richards, D.F., Bleile, R.C., Brantley, P.S., Dawson, S.A., McKinley, M.S., O'Brien, M.J.: Quicksilver: a proxy app for the monte carlo transport code mercury. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 866–873. IEEE (2017)
20. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: From performance report explorer to performance analysis tool. Procedia Computer Science **51**, 1343–1352 (2015)
21. Shende, S., Malony, A.D.: Integration and application of tau in parallel java environments. Concurrency and Computation: Practice and Experience **15**(3-5), 501–519 (2003)
22. Shende, S.S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications **20**(2), 287–311 (2006)
23. Tallent, N.R., Mellor-Crummey, J.M., Fagan, M.W.: Binary analysis for measurement and attribution of program performance. ACM Sigplan Notices **44**(6), 441–452 (2009)