Augmentation of MPI Traces Using Selective Instrumentation

Sebastian Kreutzer[1][0000−0002−1641−4342], Josep Pocurull Serra[2][0009−0004−6479−7247], Christan Iwainsky[3][0000−0002−2020−8939], Marta Garcia Gasulla[2][0000−0003−3682−9905], and Christian Bischof[1][0000−0003−2711−3032]

[1] Scientific Computing, Technische Universität Darmstadt, Darmstadt, Germany
[2] Barcelona Supercomputing Center, Barcelona, Spain
[3] Hessian Competence Center for High Performance Computing, Technische Universität Darmstadt, Darmstadt, Germany

Abstract. Tracing tools provide detailed program execution timelines aiding the identification of subtle performance issues. MPI-based tracing, as employed by Extrae, focuses on recording communication events, thus keeping the trace size manageable. However, analyzing the cause of detected issues can be difficult, as the limited information in the trace does not allow direct correlation to source code. In order to get this information, stack unwinding to a high depth may be required, which increases overhead and trace size. In this work, we present an alternative instrumentation-based approach relying on static call graph analysis. This approach identifies the functions correlating to the critical region in the trace based on the direct callers of the surrounding MPI events. The relevant call paths are then instrumented using LLVM's dynamic instrumentation feature, enabling them to be recorded in the trace. The presented method is evaluated on test cases of the OpenFOAM computational fluid dynamics solver. Results show that our approach is applicable for very large call-graphs with over 400,000 functions, while displaying moderate runtime and trace size overheads.

Keywords: Instrumentation · Profiling Tools · Static analysis.

1 Introduction

Tracing tools, such as Extrae [16] and Score-P [8], are an invaluable part of the performance analysis toolbox on HPC systems. They enable the visualization of detailed timelines of the program's parallel execution, which can help in identifying issues that may not be directly apparent in call profiles. However, traces require significantly more storage space and typically produce higher measurement overheads. Extrae addresses this concern by focusing on MPI communication and tracing only these operations by default. This allows for detailed analysis of highly-parallel runs, while keeping the generated trace sizes manageable. The drawback of this approach is that the trace provides limited information about the context of the recorded operations, apart from the communication

type and involved processes. This can lead to difficulties when analyzing such traces. Consider a scenario, in which the analysts discovers a possible performance issue located in between two MPI calls, e.g. load imbalance between ranks. The next step for the analyst is then to identify the origin of the two calls, i.e. the first common function in the two call stacks at the time of the MPI call, and find the corresponding source code region. The typical approach is to enable stack unwinding to record the call stack at each MPI event up to a certain depth. However, the minimum unwinding depth required to identify the common function cannot be determined statically and full unwinding to high depths is costly [7]. Moreover, unwinding is limited to identifying the location of the call sites only, and cannot be used to determine exact call information, such as the time at entry and exit or the value of performance counters. This information, however, may be critical to pinpoint the root of the investigated performance issue.

This work aims to aid the analyst in this task by augmenting the trace with additional call information without incurring the overhead of full instrumentation. To this end, the *Compiler-assisted Performance Instrumentation* tool (CaPI) [10,9] was extended with a dedicated selection strategy for Extrae. Based on the static call graph of the investigated program, this strategy identifies relevant call paths starting from the common caller and critical MPI calls. These call paths are then runtime-adaptable instrumented using LLVMs[11] XRay [3] feature to supplement the Extrae trace with detailed call-context information. An illustration of the proposed analysis workflow is shown in Fig. 1:

- The analyst starts with a generic low overhead trace **(1)** that contains all MPI calls and information about their direct caller.
- Detecting a performance issue, the analyst identifies the direct callers x and y of the surrounding MPI calls.
- The analyst passes a query **(2)** to the CaPI selective instrumentation tool **(4)** executing a common ancestor analysis on the static call graph **(3)** to obtain a tailored instrumentation configuration **(5)**.

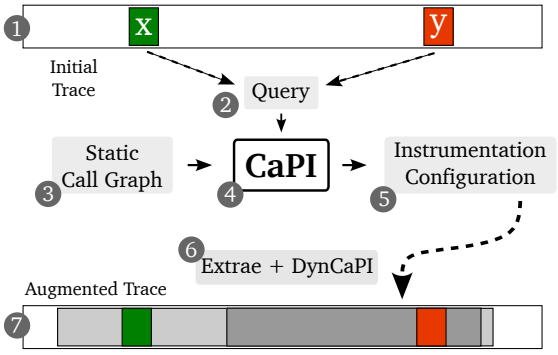

Fig. 1: Trace augmentation workflow.

– The analyst re-runs the program with Extrae and the CaPI runtime library (DynCaPI) **(6)**, generating the augmented trace **(7)**.

Our approach has several advantages over a generic unwinding approach. First, additional data is only collected for the regions of the trace relevant to the current measurement objective, thus reducing overhead and trace size. Secondly, the common origin of the investigated MPI calls is directly captured in the trace, avoiding the the manual analysis of unwound call stacks. Finally, the instrumentation offers well placed hooks to capture performance counters for the relevant region, supporting the analyst in pinpointing the investigated issue. We make the following contributions:

1. A novel approach to identify relevant common ancestors of function pairs from static whole-program call graphs.
2. An implementation of the presented static selection method within the CaPI tool, in conjunction with a dynamic filtering approach.
3. An extension of the CaPI runtime library interfacing with Extrae to trace functions instrumented via LLVM-XRay.

2 Background and Related Work

2.1 Extrae and Paraver

Extrae and Paraver [14] have been developed for performance analysis and visualization of parallel and multithreaded high-performance computing applications. They aid to understand the behavior of parallel programs to identify bottlenecks and optimize performance. Extrae is a performance profiling tool that helps capture and analyze the runtime behavior of parallel applications. It collects various types of runtime information, such as function calls, memory accesses, communication events, and synchronization points. This information is stored in trace files used for further analysis. Complementing the capabilities of Extrae, Paraver transforms the trace files generated by Extrae and provides an interactive graphical representation of the program's execution.

2.2 CaPI

Code instrumentation is a standard technique for collecting detailed performance data from programs, used by tools such as Score-P [8] and TAU [17]. The *Compiler-assisted Performance Instrumentation* tool (CaPI) was created to improve low-overhead instrumentation configurations (ICs).CaPI enables the creation of tailored ICs using its custom selection DSL. This DSL enables the analyst to combine modular *selectors* into a adjustable pipeline to fit the measurement objective and target application.

CaPI's preferred instrumentation workflow employs dynamic instrumentation based on LLVM's *XRay* feature [3]. Compared to a static instrumentation approach requiring recompilation after each change to the IC, the use of XRay greatly facilitates iterative adjustments and switching between ICs for different use cases. CaPI itself has no built-in profiling or tracing capabilities but

offers a runtime library that serves as an interface between XRay and an external measurement tool. At the time of writing, CaPI provides interfaces for Score-P, TALP [12] and a generic profiling interface compatible with GCC's `-finstrument-functions` option.

2.3 Related Work

Research specific to traces has largely focused on reducing storage requirements. CAPEK [4] is a clustering approach that applies in-situ analysis to identify groups of processes with similar performance behavior. ScalaTrace [13] applies a lossless run-time compression technique that greedily matches and combines MPI event sequences. More recently, Pilgrim [18] combines near lossless compression of MPI traces with detection of regular communication pattern. Cypress [19] incorporates static analysis, by construction a communication structure tree at compile-time in order to improve the effectiveness of the dynamic trace compression. There is little work regarding the use of instrumentation to augment existing traces. Folding [15] maps similar computation regions, e.g. solver iterations, into new synthetic regions. Metrics collected via coarse sampling are aggregated in this region, mimicking the results of high-frequency sampling. Ilsche et al. [6] investigate combining instrumentation and sampling approaches with MPI-based tracing. We are not aware of any previous work that attempts to augment specific regions of an existing trace based on the requirements of a specific analysis approach.

3 Motivating Example

This section introduces a case from OpenFOAM to illustrate the problem of correlating regions in the MPI trace with the corresponding source code locations. We examine a microbenchmark developed under the exaFOAM project [4], simulating a rotating car wheel. Performance analysis of the case revealed a region displaying load imbalance between ranks, shown in Fig. 2. The primary issue is that certain ranks have excessively long outside MPI execution times, causing all other ranks to idle at the `MPI_Bcast` call. To investigate this issue, the performance analyst needs to identify the code location corresponding to this outside MPI state. With the current state of the art, this can be achieved by enabling stack unwinding in Extrae and manually identifying the lowest common caller in the call stack of the encapsulating `MPI_Waitall` and `MPI_Gather` calls. In this case, the common caller is in the tenth level of the `MPI_Waitall` call stack and in the sixth level of the `MPI_Gather` call stack, as shown in Fig. 3. However, unwinding to this depth increased the runtime by 19% and trace size by 133%. The aim of this work is to improve this process by directly tracing relevant call events, thereby eliminating the need for manual examination of the call stack and providing additional performance data within the selected region.

[4] https://exafoam.eu

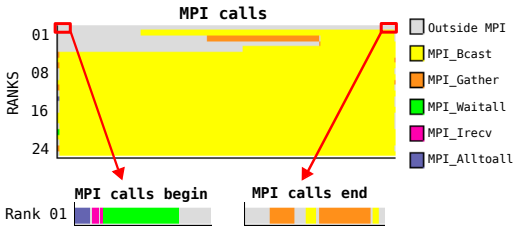

Fig. 2: Region of the Paraver trace showing MPI calls

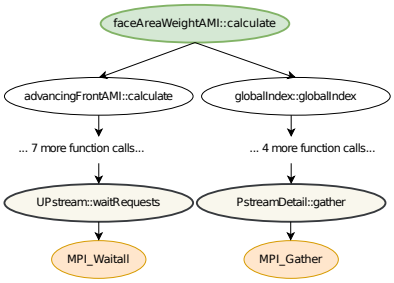

Fig. 3: Call paths corresponding to the critical trace region

4 Static Selection and Instrumentation

This section describes the problem of identifying the common caller and outlines our novel static selection approach.

4.1 Graph Definitions

Different graph representations of the call hierarchy are used at different points of the program's lifetime. We use the following definitions:

Static call graph: The static call graph G_s is determined analytically from the source or other code representation. It contains all function calls that are statically present in the code. Since it does not take program inputs into account, it may contain edges that are never taken during execution.

Dynamic call graph: The dynamic call graph G_d represents the call hierarchy of the program over the runtime of a specific execution. It is a subgraph of the static call graph for a given set of input parameters.

Dynamic call path: The dynamic call path to a function v_n is given by the vertex sequence $p = (v_1, ..., v_n)$, where v_1 is a vertex with in-degree zero (e.g. the `main` function). It is equivalent to the call stack at the time of function entry.

4.2 Lowest Common ancestor problem

Given the dynamic call paths p_x and p_y of two function calls to x and y, the wanted function is the lowest common ancestor [2] of the union of the two call paths. We refer to this node as the *dynamic lowest common ancestor* $dLCA(p_x, p_y)$. Fig. 4 provides an example. However, p_x and p_y are not available in the original trace. Hence, our objetive for the trace augmentation is the following: Given only the delimiting target functions x and y, identify the $dLCA$ for a region in the trace corresponding to call paths p_x and p_y. By instrumenting the $dLCA$ and the subjacent call paths, the analyst can subsequently identify the corresponding source code regions.

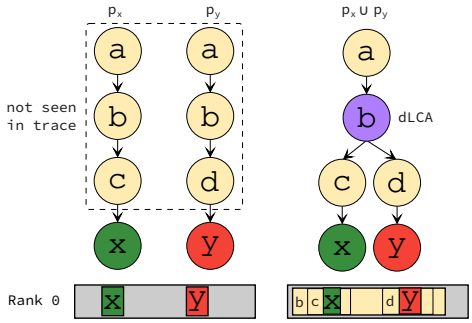

Fig. 4: Visualization of the dLCA of two call paths. The left side shows the two call paths p_x and p_y, where x and y correspond to the two MPI calls observed in the trace (shown below). The right side shows the union of these paths, with b as the resulting $dLCA$. By instrumenting b and the callees in the subtree, the trace can be augmented as shown.

4.3 Static analysis approach

In our approach, we rely on the static call graph to determine potential candidates for the $dLCA$. There is generally more than one possible $dLCA$, as illustrated in Fig. 5. Our aim is therefore to identify a subset of nodes, called

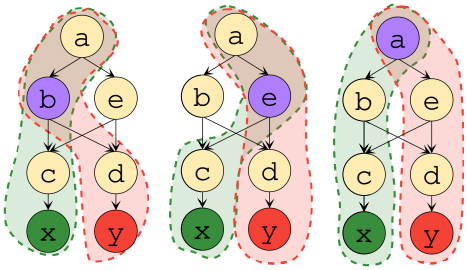

Fig. 5: This figure shows a possible static call graph corresponding to the previous call path example. Due to the additional call edges, multiple paths to x and y with different dLCAs are feasible. Here, a, b and e are all possible candidates.

candidates, that comprise the set of possible dLCAs for all paths to x and y. More formally, we define candidates as follows:

Definition 1 (Candidate). *Node $v \in CA(x, y)$ is a* candidate *in the static call graph, if there are paths p_x and p_y such that $v = dLCA(p_x, p_y)$.*

For complex codes with many candidates, instrumenting all possible candidates is too costly. Thus, we heuristically (see 4.5) select a subset of candidates and corresponding call paths for instrumentation.

4.4 Identifying Candidates

We use *postdominance* to identify candidates: Node v is said to *post-dominate* w w.r.t. exit node e if all paths from v to e must pass through u. It can be shown that $v \in CA(x, y)$ is a *candidate* in the static call graph iff there is no node $w \neq v$ that post-dominates v w.r.t. x and y. Hence, candidates can be identified using the following steps (illustrated in Fig. 6):

1. Perform a breadth-first search to find the common ancestors of x and y.
2. Compute post-dominators of all common ancestors w.r.t. x and y, e.g. by using a worklist algorithm to solve the underlying data flow problem [5].
3. Starting with the lowest common ancestors, traverse the graph bottom-up to find intersections between post-dominators w.r.t. x and y. Mark nodes as candidates if the intersection contains only the node itself.

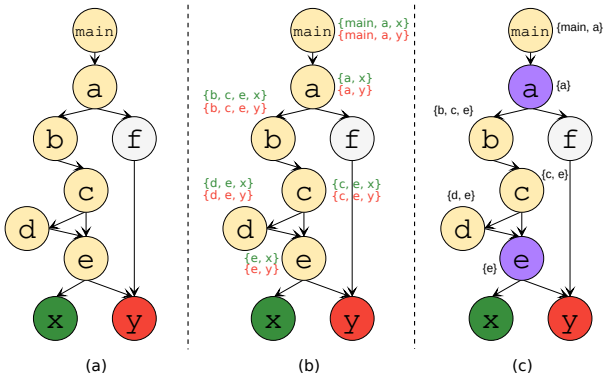

Fig. 6: Example for the candidate selection process. In (a), the common ancestors of x and y are highlighted in yellow. In (b), the post-dominators w.r.t. x (green) and y (red) are computed for each common ancestor, displayed next to the nodes. Finally, (c) depicts the resulting candidates a and e.

4.5 Heuristic Filtering

To reduce the number of candidates, we focus on candidates that are structurally closest to the two target functions. To this end, we examine the possible paths from the candidate node to x and y. We call candidates *distinct*, if there are paths to x and y that do not pass through another candidate. We call them *partially distinct* if this criterion is fulfilled for only one of the target functions. In Fig. 6, for example, e is a distinct candidate and a is partially distinct. We explore two selection heuristics: a *minimal selection* that considers only distinct candidates and an *extended selection* that includes partially distinct candidates as well. *Minimal selection* results in a smaller IC and fewer recorded events, but might fail to capture some calls to x and y. In Fig. 6, for example, the call path $p_y = (main, a, f, y)$ is not recorded with *minimal selection*. *Extended selection* ensures that for each call to x and y, the preceding call path is at least partially recorded, even if the dLCA itself is not instrumented.

4.6 Handling Cycles

The presented selection mechanism was designed for DAGs. In general, the static call graph of a program cannot be assumed to be acyclic, e.g. due to recursion or over-approximation. This issue can be circumvented by using a technique called *condensation*, turning a directed graph G into a DAG by combining the nodes of each *strongly connected component* (SCC) into a supervertex. The nodes of the resulting graph G_{SCC} are the SCCs of G, and (C_1, C_2) is an edge, iff $\exists u \in C_1, v \in C_2$ such that (u, v) is an edge in G. The presented algorithm can then be applied to this cycle-free condensation graph, instrumenting all subvertices of selected SCCs.

5 CaPI Extension for Extrae

The CaPI runtime library (*DynCaPI*) was extended to record instrumented functions in the Extrae trace.

Extrae Interface: In order to use CaPI in conjunction with Extrae, the new Extrae variant of the DynCaPI library is statically linked into the target binary. Additionally, the Extrae library needs to be linked statically or using `LD_PRELOAD`. If DynCaPI detects the Extrae API, the available XRay sleds are collected and cross-checked with the user-specified IC file. Included functions are then patched via XRay and registered with Extrae. We are currently using a "flat" tracing mode that records nested calls on a single timeline, always displaying the function on top of the call stack. The runtime library maintains a simple shadow stack to correctly reset the current function in the trace on exit events.

Scoped Instrumentation: The common ancestor IC includes not only each selected candidate dLCA, but also all functions on call paths below it to any of the two target functions. These functions may also be called in different, unrelated, parts of the program, potentially leading to a lot of unwanted trace events. To combat this, CaPI was extended with the notion of *scope triggers*. Function events are only recorded in the trace, if a scope trigger is part of the current call stack. By marking candidates as scope triggers, we are able to filter out instrumented function invocations from unrelated program parts.

6 Evaluation

The presented selection mechanism is evaluated on the `lulesh` proxy application [1], as well as two much larger OpenFOAM benchmarks using the `icoFoam` and `pimpleFoam` solvers. The `pimpleFoam` setup is the benchmark presented in Section 3. For each of the benchmarks, we first generated an MPI-based trace with Extrae. We then identified interesting regions in the trace, alongside the direct callers x and y of the surrounding MPI calls. The corresponding dLCA was then manually located from the call paths collected via stack unwinding.

Details for the setups are shown in Table 1. Measurements were conducted on the CLAIX-2018 cluster[5], running on Intel Xeon Platinum 8160 CPUs.

Table 1: Benchmark details. `#f` is the number of functions in the static call graph. The `depth` is the call-depth distance to reach x and y from `dLCA`.

Benchmark	#f	x	y	dLCA	depth
`lulesh`	3,360	`CommRecv`	`CommSend`	`LagrangeLeapFrog`	1
`icoFoam`	496,959	`waitRequests`	`allReduce<bool>`	`main`	5
`pimpleFoam`	496,964	`waitRequests`	`gather`	`calculate`	10

6.1 Building the Call Graph

The whole-program call graph for `pimpleFoam` consists of 496,964 function nodes. The call graph for `icoFoam` is of similar size as it uses the same library set. CaPI's SCC analysis shows that 99.94% of the contained SCCs are acyclic, i.e. of size 1. The remaining 276 cyclic SCCs contain a total of 1984 functions, of which 562 make up the largest SCC. A manual examination of this SCC showed that the concerned functions are mostly related to error handling and logging. Such cycles in the static call graph are likely a result of over-approximation of potential callers and therefore not expected to materialize during execution. The `lulesh` call graph is cycle-free.

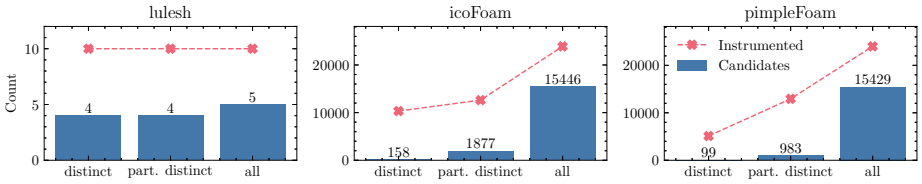

Fig. 7: Number of selected candidates and corresponding instrumented functions.

6.2 Selection

For each setup, CaPI queries were run to find candidates for the target functions. Fig. 7 shows the number of candidates and resulting number of instrumented functions. For `lulesh`, the target functions are called in a single code location. This results in a very small number of candidates as well as instrumented functions. There is no difference between the minimal and extended selection heuristics. The OpenFOAM code base is much more complex and uses the target functions in different contexts, resulting in a much higher number of candidates. One observation can be made regarding the change in number of

[5] https://help.itc.rwth-aachen.de/en/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/

instrumented function when increasing the number of selected candidates. There are approx. 10 times as many partially distinct compared to distinct candidates for `pimpleFoam`. However, the number of instrumented functions is only twice as high. This is due to large parts of the instrumented call paths already being covered by the distinct candidate instrumentation. A similar effect can be observed when going from partially distinct to all candidates.

6.3 Tracing

We executed the benchmarks in different configurations, comparing the standard Extrae trace with and without unwinding to the *minimal* and *extended* variants of our instrumentation approach. The results are shown in Table 2. We also considered a purely dynamic augmentation mode, in which full instrumentation is turned on between calls to x and y. This, however, increased the runtime too much to be feasible, and is therefore not included here. Augmenting the trace using the *minimal* heuristic was sufficient to cover the investigated trace regions in all but one cases and kept both runtime and trace size overheads below 8%. The `icoFoam` case was the only one requiring the *extended* heuristic, increasing runtime by 15% and trace size by 10%. The traditional unwinding approach introduced comparable runtime overheads to the *extended* heuristic but yielded much larger traces.

7 Conclusion

We present a method for augmenting MPI traces with call contexts using selective instrumentation based on static call graph analysis. Evaluation on two

Table 2: Trace results comparing the two selection heuristics with base Extrae and stack unwinding to a depth of 10. T_{init} is CaPI initialization and XRay patching time, T_{total} the total run time in seconds. Trace size is measured in MB. *dLCA coverage* states whether the configuration recorded events for the investigated region.

Benchmark	Variant	T_{init}	T_{main}	Trace Size	dLCA coverage
`lulesh`	`extrae`		104	97	-
	`unwind_10`		105 (+1%)	178 (+92%)	-
	`capi_min/ext`	1.7	104 (+0%)	105 (+8%)	✓
`icoFoam`	`extrae`		95	285	-
	`unwind_10`		125 (+31%)	615 (+118%)	-
	`capi_min`	4	98 (+3%)	287 (+2%)	✗
	`capi_ext`	4	109 (+15%)	315 (+10%)	✓
`pimpleFoam`	`extrae`		283	9,227	-
	`unwind_10`		337 (+19%)	21,507 (+133%)	-
	`capi_min`	10	300 (+6%)	9,729 (+5%)	✓
	`capi_ext`	12	364 (+28%)	13,425 (+38%)	✓

OpenFOAM test cases shows that our static selection method is viable even for very large applications with call graphs consisting of hundreds of thousands function nodes. The regions of interest were successfully instrumented, extending the trace with detailed function call information and simplifying the process of mapping detected performance issues back to the responsible source code locations. The extended selection heuristic in particular was able to cover the region of interest in all cases, making it a viable alternative to stack unwinding, which yielded comparable runtime overheads but much larger traces, while providing less detailed information to the analyst. Improvements can be made by streamlining the process of generating the whole-program static call graph, which currently requires some initial configuration effort. Furthermore, the accuracy of the selection may be improved by reducing the amount of over-approximated edges in the call graph, e.g. by incorporating analysis of the intermediate representation during compilation.

Acknowledgments. This research has been conducted as part of the exaFOAM Project https://www.exafoam.eu, which has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 956416. The JU receives support from the European Unions Horizon 2020 research and innovation programme and France, Germany, Italy, Croatia, Spain, Greece and Portugal. Furthermore, this work was funded by the Bundesministeriums für Bildung und Forschung (BMBF) - 16HPC023.

The authors gratefully acknowledge the computing time provided to them at the NHR Centers NHR4CES at RWTH Aachen University (project number p0021597) and TU Darmstadt. This is funded by the Federal Ministry of Education and Research, and the state governments participating on the basis of the resolutions of the GWK for national high performance computing at universities (www.nhr-verein.de/unsere-partner).

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Tech. Rep. LLNL-TR-490254
2. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms **57**(2), 75–94 (2005). https://doi.org/https://doi.org/10.1016/j.jalgor.2005.08.001, https://www.sciencedirect.com/science/article/pii/S0196677405000854
3. Berris, D.M., Veitch, A., Heintze, N., Anderson, E., Wang, N.: XRay: A function call tracing system (2016), https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45287.pdf
4. Gamblin, T., de Supinski, B.R., Schulz, M., Fowler, R., Reed, D.A.: Clustering performance data efficiently at massive scales. In: Proceedings of the 24th ACM International Conference on Supercomputing. p. 243252. ICS '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1810085.1810119, https://doi.org/10.1145/1810085.1810119
5. Hecht, M.S., Ullman, J.D.: A simple algorithm for global data flow analysis problems. SIAM Journal on Computing **4**(4), 519–532 (1975). https://doi.org/10.1137/0204044

6. Ilsche, T., Schuchart, J., Schöne, R., Hackenberg, D.: Combining instrumentation and sampling for trace-based application performance analysis. In: Niethammer, C., Gracia, J., Knüpfer, A., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2014. pp. 123–136. Springer International Publishing, Cham (2015)
7. Iwainsky, C., Lehr, J.P., Bischof, C.: Compiler supported sampling through minimalistic instrumentation. In: 2014 43rd International Conference on Parallel Processing Workshops. pp. 166–175 (2014). https://doi.org/10.1109/ICPPW.2014.33
8. Knüpfer, A., Rössel, C., An Mey, D., 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. Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing 2011 pp. 79–91 (2012). https://doi.org/10.1007/978-3-642-31476-6_7
9. Kreutzer, S., Iwainsky, C., Garcia-Gasulla, M., Lopez, V., Bischof, C.: Runtime-adaptable selective performance instrumentation. In: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 423–432. IEEE Computer Society, Los Alamitos, CA, USA (may 2023). https://doi.org/10.1109/IPDPSW59300.2023.00073, https://doi.ieeecomputersociety.org/10.1109/IPDPSW59300.2023.00073
10. Kreutzer, S., Iwainsky, C., Lehr, J.P., Bischof, C.: Compiler-assisted instrumentation selection forălarge-scale C++ codes. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) High Performance Computing. ISC High Performance 2022 International Workshops. pp. 5–19. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-23220-6_1
11. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis amp; transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665
12. Lopez, V., Ramirez Miranda, G., Garcia-Gasulla, M.: Talp: A lightweight tool to unveil parallel efficiency of large-scale executions. In: Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy. p. 310. PERMAVOST '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3452412.3462753, https://doi.org/10.1145/3452412.3462753
13. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: Scalatrace: Scalable compression and replay of communication traces for high-performance computing. Journal of Parallel and Distributed Computing **69**(8), 696–710 (2009). https://doi.org/https://doi.org/10.1016/j.jpdc.2008.09.001, https://www.sciencedirect.com/science/article/pii/S074373150800169X, best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)
14. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31 (1995)
15. Servat, H., Llort, G., Giménez, J., Huck, K., Labarta, J.: Folding: Detailed analysis with coarse sampling. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011. pp. 105–118. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

16. Servat, H., Llort, G., Huck, K., Giménez, J., Labarta, J.: Framework for a productive performance optimization. Parallel Computing **39**(8), 336–353 (2013)
17. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. International Journal of High Performance Computing Applications **20**(2), 287–311 (may 2006). https://doi.org/10.1177/1094342006064482, https://dl.acm.org/doi/10.1177/1094342006064482
18. Wang, C., Balaji, P., Snir, M.: Pilgrim: Scalable and (near) lossless mpi tracing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476151, https://doi.org/10.1145/3458817.3476151
19. Zhai, J., Hu, J., Tang, X., Ma, X., Chen, W.: Cypress: Combining static and dynamic analysis for top-down communication trace compression. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 143–153 (2014). https://doi.org/10.1109/SC.2014.17