Binary Instrumentation for Scalable Performance Measurement of OpenMP Applications

Julien JAEGER [a,c], Peter PHILIPPEN [b], Eric PETIT [a], Andres CHARIF RUBIAL [a,d], Christian RÖSSEL [b], William JALBY [a,d], and Bernd MOHR [b]

[a] *PRISM, University of Versailles, France*
[b] *Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany*
[c] *CEA, DAM, DIF, F-91297, Arpajon, France*
[d] *Exascale Computing Research, France*

Abstract.
In this paper we present a binary instrumentation methodology to monitor runtime events. We demonstrate our approach on OpenMP constructs for the Intel and GNU compilers. A binary-level static analysis detects the compiler patterns and the runtime function calls corresponding to OpenMP regions. To this effect we integrate the software tool MAQAO with the scalable measurement infrastructure Score-P. We design a new interface and modify both tools to support the new events. The main advantages of using binary instrumentation are the possibility to retrieve implicit runtime events, to instrument without recompilation, to be independent from the language, and not to interact with compiler optimization. Our validation experiments and first results shows that binary instrumentation has not introduced any additional overhead.

Keywords. Binary instrumentation, MAQAO, Score-P, OpenMP, performance, measurement, analysis, trace, profile

Introduction

Dynamic performance analysis is essential to optimize applications. To capture the complete behavior of an application, a widely used method is probe insertion combined with profiling and tracing. While tracing gives very detailed insight, dynamic events can easily occur billions of time, thus increasing time and memory overhead. In extreme cases the analysis cannot be done in a reasonable time. With the growing size of both systems and applications, this problem gets worse and minimizing the impact of instrumentation becomes paramount.

In our work, we concentrate on probe insertions around program structures such as functions, loops, and OpenMP runtime events. Traditional instrumentation approaches are based on source or compiler-level instrumentation, or library interposition. The two former ones respectively imply a strong dependency on language and compilers. It limits the portability of the tools. Library interposition [1,2] implies a strong commitment of

system and runtime developers. Specification efforts, such as the OMPT interface [2] still need to be implemented in a future OpenMP release.

In this paper, we use binary instrumentation with MAQAO to insert the probes directly into the application binaries. This approach is language agnostic but is limited to one particular instruction set, x86_64 in our case. It has no interaction with the build process. It does not need any recompilation of the target application. It offers the possibility to detect and instrument implicit runtime events in the binary such as the implicit barrier after a `parallel for` region in OpenMP. Contrary to source-level instrumentation, it has no interaction with the compiler optimization pass. We integrated this approach with the open-source project Score-P [3]. It provides a portable measurement infrastructure and enables automatic instrumentation and measurements, with traces and profiles written in the open data formats OTF2 [4] and CUBE4 [5]. Therefore, the generated traces and profiles are compatible with other existing tools operating on these formats. The main contributions presented in this paper are:

- a methodology to instrument hidden runtime events in a binary
- the management of undeterministic events closure at runtime
- the integration of the binary instrumenter MAQAO with the Score-P measurement infrastructure

Sections 1 and 2 describe the MAQAO and Score-P tools. Section 3 presents some related work about program instrumentation. Section 4 describes the methodology employed for detecting the runtime events in a binary and the runtime support to handle it. Section 5 presents our results on several benchmarks and we finally conclude on results and future work.

1. Score-P

The measurement system Score-P [3] is a community project, initiated by Vampir [6], Scalasca [7], Tau [8] and Periscope [9] tools, to reduce redundant efforts in maintenance, development, support and training. Score-P performs automatic instrumentation of applications, employing compiler and source-to-source instrumentation as well as library interposition. The supported programming paradigms are MPI, OpenMP and hybrid combinations. Since the H4H project [10] (Hybrid programming for Heterogeneous architectures) Score-P supports accelerators such as GPUs [11].

Figure 1 shows the underlying architecture of Score-P. The different instrumenters are shown at the bottom. The measurement infrastructure is responsible for generating timestamps for events and recording profiles and traces. It is linked to the user's application via the adapters. Since different instrumentation tools can be used, the adapters are necessary to translate the specific tools interface to Score-P requirements.

The results are stored in OTF2 traces [4] or CUBE4 [5] profiles. The CUBE4 format stores profiles in aggregated form along three dimensions: metrics, call-tree and system-tree. In an OTF2 trace, every entry and exit of a region are stored explicitly, resulting in a greater demand of memory and storage on disk. The data is distributed over several files and consists of region definitions and events specifying the entries and exits of these regions.

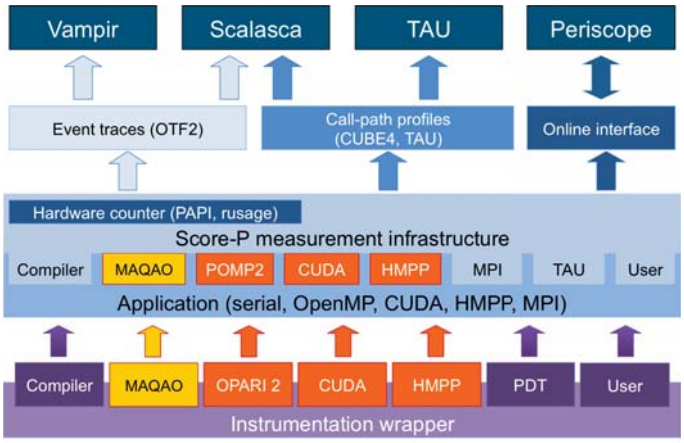


Figure 1. Overview of the Score-P architecture. Extensions during the H4H project are highlighted, with special focus on the MAQAO integration.

2. MAQAO

MAQAO (Modular Assembly Quality Analyzer and Optimizer) analyzes binary codes and provides application developers with reports to optimize their code. The tool mixes both static code quality evaluation [12], and dynamic profiling and characterization [13]. This is based on the ability to reconstruct low-level and high-level structures, such as basic blocks, loops, functions, and call-sites. Another main feature of MAQAO is its extensibility. Users can easily write their own plug-ins, using the embedded scripting language Lua. It allows fast prototyping of new tools based on MAQAO.

MAQAO also provides a domain-specific language for binary instrumentation: MIL (MAQAO Instrumentation Language) [14]. It is an event-driven and object-oriented language. The objective is to provide a high-level view of objects that can be manipulated by events. It controls the insertion of probes and seamlessly produces the corresponding modified binary. For instance the loop object exposes entries, exits and back-edge events (i.e. the edge that close the loop). To count the numbers of iterations of all loops, a probe can be added on the back-edge event.

In general, in the MIL file, the application developers subscribe to events and can define the following elements:

- Filters: A means to select a specific subset of objects to process.
- Actions: Additional processing on the object regardless of probe insertion.
- Probes: A probe is defined by its function name, the name of the dynamic library in which it is contained and the list of its parameters.

3. Related Work

To assist developers, there are a number of performance analysis tools with different approaches for different purposes. One of the main differences between these tools, lies

in the used instrumentation methods: source-to-source, compiler, binary instrumentation and library interposition.

Source-to-source instrumentation: This is the state-of-the-art instrumentation method for OpenMP such as in OPARI2 [15] or ompP [16]. However, it requires re-compiling and relinking of the code and the inserted probes influence the compiler optimization. Furthermore, some internal runtime events, such as implicit barriers cannot be instrumented at source-level.

Compiler instrumentation: Compilers, such as `ICC` [17], provide function and loop instrumentation. However it cannot be fine tuned and can prevents some optimization. It results in large measurement overhead when the probes are inserted for frequent events with little workload.

Library interposition: Library functions are intercepted and information about their usage is collected. Then the call is passed to the actual library implementation. This is the standard method for measuring MPI communication. As an OpenMP directive gets translated into a certain combination of calls, this method could only be used to instrument the OpenMP runtime library functions.

Binary instrumentation: There are multiple solutions for binary instrumentation such as Intel PIN [18], Dynamorio [19] or dyninst [20]. The first two are powerful and robust, but their dynamic overhead limits them to measurements that are not related to time. Dyninst on the other side is a dynamic binary patcher. This versatile solution proves to be efficient in handling simple cases, but fails for complex program structures. With Cobi [21], it has been integrated into Score-P to instrument functions. However, dyninst cannot perform the necessary analysis to instrument events with high-level semantics, such as runtime events.

4. Software Extensions

The integration of MAQAO and Score-P requires the definition and implementation of a new interface and of extensions in both tools. In this section we present the common interface, the runtime instrumentation methodology with MAQAO and the integration with Score-P.

4.1. Common Interface

The MAQAO requirements, and the information it can provide, are different from the other tools already embedded in Score-P. Therefore we cannot reuse the existing interfaces to Score-P. We propose two new interfaces.

The Score-P-MAQAO user INTerface, SMINT, defines probe functions for user loops and functions. It registers instrumented code regions to the measurement system with a unique id. This id encodes all necessary information about the name of the functions, the source code location and some auxiliary data.

The Score-P-MAQAO user interface for OpenMP, SMOMP, defines specific probe functions for OpenMP regions. It also provides specific probes function to initialize the thread-safe data structures, which can then be accessed by all threads during the parallel execution.

Original code	MAQAO instrumented without debug info	
	with ICC runtime	with GCC runtime

`#pragma omp parallel`
...
`}`
	SMOMP_Parallel_fork()
	`#pragma omp parallel`
	`{ SMOMP_Parallel_begin()`
	...
	`SMOMP_Parallel_end() }`
	SMOMP_Parallel_join()
		SMOMP_Parallel_fork()
		`#pragma omp parallel`
		`{ SMOMP_Parallel_begin()`
		...
		`SMOMP_Parallel_end()}`
		SMOMP_Parallel_join()

`#pragma omp for nowait`
`{ for(...) {....} }`
	SMOMP_For_nowait_enter()
	`#pragma omp for nowait`
	`{ for(...) {....} }`
	SMOMP_For_nowait_exit()
		SMOMP_For_nowait_enter()
		`#pragma omp for nowait`
		`{ for(...) {....} }`
		SMOMP_For_nowait_exit()

`#pragma omp for ordered`
`{ for(...) {....} }`
	SMOMP_For_ordered_nowait_enter()
	`#pragma omp for ordered`
	`{ for(...) {....} }`
	SMOMP_For_ordered_nowait_exit()
		SMOMP_For_ordered_nowait_enter()
		`#pragma omp for ordered`
		`{ for(...) {....} }`
		SMOMP_For_ordered_nowait_exit()

`#pragma omp sections`
`{`
`#pragma omp section`
`{...}`
`}`
	SMOMP_Sections_enter()
	`#pragma omp sections`
	`{`
	`#pragma omp section`
	`{ SMOMP_Section_begin()`
	...
	`SMOMP_Section_end() }`
	SMOMP_Sections_exit()
		SMOMP_Sections_enter()
		`#pragma omp sections`
		`{`
		`#pragma omp section`
		`{` **SMOMP_Section_begin()**
		...
		SMOMP_Section_end()}
		SMOMP_Sections_exit()

`#pragma omp master`
`{ ... }`
	SMOMP_Master_enter()
	`#pragma omp master`
	`{ ... }`
	SMOMP_Master_exit()
		SMOMP_Master_enter()
		`#pragma omp master`
		`{ ... }`
		SMOMP_Master_exit()

`#pragma omp single`[1]
`{ ... }`
	SMOMP_Single_enter()
	`#pragma omp single`
	`{ ... }`
	SMOMP_Single_exit()
		SMOMP_Single_enter()
		`#pragma omp single`
		`{ ... }`
		SMOMP_Single_exit()

`#pragma omp barrier`[2]	SMOMP_Barrier_enter() `#pragma omp barrier` SMOMP_Barrier_exit()	SMOMP_Barrier_enter() `#pragma omp barrier` SMOMP_Barrier_exit()
`#pragma omp atomic` ...	SMOMP_Atomic_enter() `#pragma omp atomic` ... SMOMP_Atomic_exit()	SMOMP_Atomic_enter() `#pragma omp atomic` ... SMOMP_Atomic_exit()
`#pragma omp task` `{...}`	SMOMP_Task_enter() `#pragma omp task` `{` **SMOMP_Task_begin()** ... **SMOMP_Task_end()** } **SMOMP_Task_exit()**	SMOMP_Task_enter() `#pragma omp task` {SMOMP_Task_begin() ... **SMOMP_Task_end()** } **SMOMP_Task_exit()**

[1] works also for `#pragma omp critical` and `#pragma omp ordered`
[2] works also for `#pragma omp flush` and `#pragma omp taskwait`

Figure 2. Recognized pragmas. Probe instertions printed in bold are not yet supported by the current implementation.

4.2. MAQAO Runtime Instrumentation

MAQAO instruments the program by inserting probes into the binary. At binary level, the runtime constructs are inserted as assembly macros, runtime calls or structural modification. In this section, we present the methodology to reconstruct the source-level runtime events and apply it on the Intel and GNU OpenMP runtimes.

4.2.1. General Methodology

High-level runtime semantics are translated in various forms into the binary. We classify them into three categories which can appear in combination. For each category, we propose techniques to reconstruct the information.

- **Simple call-sites:** Events of this class translate into one or multiple well-defined runtime function calls. To detect them, we parse the instructions and function symbols.
- **Inserted assembly macro:** Runtime events may be translated into compiler assembly macros. These can either be a sequence of assembly instructions or even inlined functions. The code does not vary with the context, therefore a simple pattern matching is sufficient.
- **Structural modification:** This is the most complex case. The events are translated into structural modifications of the user code such as outlining or external loop-level addition. To retrieve the information we must perform static analysis on the Control Flow Graph of the program.

Our methodology to determine the source-level runtime directives is based on analyzing simple unitary benchmarks with source-level instrumentation. After compilation, we disassemble it with MAQAO and manually analyze the assembly code surrounding the source-level probes. For OpenMP this is done with the HLRS unitary OpenMP benchmark [22] suite and OPARI2.

4.2.2. Implementation of the OpenMP Instrumentation

In this section, we study the runtime calls inserted by the `ICC` and `GCC` compilers, respectively the *KMPC* and *GOMP* runtimes. The two compilers use different methods and while most of the directives are easily recognized, some are very difficult to identify. The table in figure 2 shows the probe insertion scheme for `ICC` and `GCC`. The bold events are the ones we currently cannot recognize. The remainder of this section describes some examples for simple and complex detection cases.

Simple cases: Most of the directives have well defined bounds, either by runtime calls or particular patterns of instructions. If we consider the `omp parallel` directive, we can observe two cases: with ICC, a runtime call defines the boundaries of the parallel directive, while for GCC the parallel region is outlined in a new function. To instrument around this directive, the measurement functions have to be inserted around the ICC runtime function call or the GCC outlined function.

For some OpenMP directives, the runtime call insertions are the same and cannot be distinguished. Therefore it produces ambiguous results. For example, with ICC, any barrier will be represented with a specific runtime call. It is not possible to distinguish explicit user barriers from a call insertion due to an implicit barrier (e.g. following an `omp for` directive). In that case we decide to consider all barriers as explicit. Users can easily interpret the results from the source code thanks to the compiler debug information.

Complex cases: When the bounds are not explicitly translated into a specific syntactic unit, we have to consider using static analysis as stated in section 4.2.1. These cases are illustrated by the two following representative cases.

For the `omp for` directive with the `ordered` clause, ICC generates three different runtime calls. The runtime checks several times if the current iteration of the parallel

loop is the last one, and which thread should handle the next iterations. After the first runtime call at the beginning of the loop, a branch generates two execution paths. The first branch is the subgraph containing the loop, and the second branch is a bypass to the end of the loop. While the entry probe function for the region is inserted at the beginning, the closing measurement function must be inserted after the join of the two branches. To find the correct block, we search the *post-dominator* block of the beginning block to insert our exit probe.

For `omp for` with GCC, no runtime function call is inserted. At the beginning of a parallel region, we identify a specific pattern of instructions that is inserted to compute the work per thread. Starting from this pattern, we follow the CFG path to find the first diverging point. This diverging point is the GCC equivalent of the fork node of the previous example on ICC. We can therefore apply the same method to exhibit the end of the directive by searching the post-dominator of the diverging node.

4.2.3. Interaction with compiler optimizations

Using source-to-source instrumentation can impact compiler optimization. Indeed, the insertion of calls to external functions can prevent some optimization such as inlining or loop fusion, modify the result of compiler heuristics, or impact the CFG structure and the binary layout such as the code alignment. Binary instrumentation does not. We observe many of these interactions. The following example illustrates the impact on the loop fusion optimization.

In some generated binaries, the parallel region can cover several loops which are not in the same directive in the source code. When instrumenting with OPARI2, the measurement functions are inserted around each parallel loop. When the code is not instrumented, the compiler uses the same blocks to control the two loops. When a source-level probe call is inserted between the loops, the compiler must respect the semantic and since it has no information if this call has side effects, it must finish the first loop before starting the second one. Therefore, it cannot do the fusion anymore. Furthermore, with OPARI2, even if there is no loop iteration computed, the measurement still takes place. To mimic the behavior of the source-level instrumentation, we need to insert probes in the bypassing branch for each loop. For multiple parallel loops we take care to instrument each of them to remain coherent with OPARI2 instrumentation.

4.3. Score-P

The modular design of the Score-P architecture allows to easily add extensions. There are two main challenges in integrating the new MAQAO adapter. The first lies in mapping MAQAO data to the thread-safe infrastructure in Score-P. The second is to keep track of code regions with un-deterministic exit events due to the limitation of binary level instrumentation.

4.3.1. Ensuring Thread-Safe Data Accesses in the Measurement Core

During the measurement, the probes call the adapter wrappers which activate the measurement core to store the information. In a multi-threaded application, we must maintain distinct memory spaces for the data measured by each thread.

The OPARI2 source-to-source instrumentation solves this issue by using OpenMP `threadprivate` data. The compiler is responsible for creating the necessary OpenMP

thread-safe data structures. This approach is not possible with MAQAO as the relevant directives would have to be inserted before compilation. However there are OpenMP runtime functions to determine the level of nesting of parallel regions and to query the thread IDs of each ancestor thread. For the integration, these functions are used to store the nesting hierarchy for all threads in a tree structure and associate distinct spaces in memory with each node in the tree.

4.3.2. Keeping Track of Un-Deterministic Exit Events

Score-P requires that entry and exit events are mapped to each other in a perfectly nested way. However, during binary instrumentation with MAQAO, some exits of regions cannot be associated with a specific entry.

To circumvent this restriction we propose a methodology, based on stack, to ensure the consistency of the measurement. This approach is implemented in the MAQAO-Score-P adapter. Since all events are perfectly nested and not cross over, when an exit event occurs, it should close the last opened event. Therefore, we store all enter events in a stack and ensure the consistency of the exits. If an un-deterministic exit event occurs we close the last open region on the stack. If an exit event occurs that has an identifier which differs from the top of the stack, all events are closed until the corresponding event in the stack is found. Even if this method introduces some inaccuracy in the time measurement, we are able to emit a warning on the concerned events and the rest of the measurement is not corrupted.

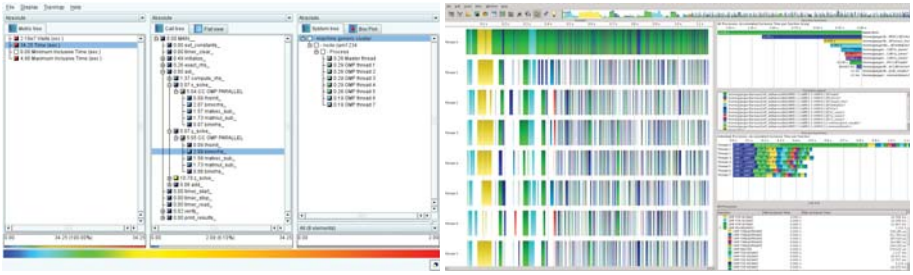
5. Experiments

The current software prototype has been successfully installed and tested on the BULL Nova Cluster provided for the H4H project. MAQAO and Score-P are both available in the *module* environment. We tested our software with ICC 12.1.0 and 12.1.5 and on GCC 4.2.1 and 4.6.3. To validate our approach, we verified our instrumentation on the OpenMP validation suite of HLRS [22]. This suite is intended to evaluate the robustness of the OpenMP implementations of compilers; indeed it extensively tests different combinations of directives. We used the NAS parallel benchmarks to validate our results and compare to the source-to-source instrumenter OPARI2.

Except for OpenMP tasks, all OpenMP constructs can be instrumented with the methodology presented in section 4.2. For tasks, the methodology fails to find where to insert the probes as no runtime calls, structural changes or specific instruction patterns have been identified. The Score-P measurement infrastructure is only able to measure tied tasks. For untied tasks the problem lies in finding the potential migration point. For this problem, no solution has yet been found.

For the implemented events, the instrumentation produces traces and profiles that are similar to the ones based on OPARI2 instrumentation. The measurements are consistent for Score-P and can therefore generate valid output. The results can be examined with Vampir and CUBE. Figure 3(a) shows an example for a profile in CUBE and figure 3(b) shows a trace in Vampir.

For some NAS benchmarks the internal thread safe memory management of Score-P failed on our setup, either using MAQAO or OPARI2 instrumentation. These problems will be solved in the next release of Score-P.



(a) Cube profile view (b) Vampir trace view

Figure 3. Views of the NAS BTbenchmark trace and profile generated with MAQAO-Score-P

The instrumented binary performance is similar to the OPARI2 source level instrumented application. However tracing all loops and functions can be prohibitive. To solve this problem without sampling, it is necessary to instrument only a subset of specific loops and functions. To this end, our tool proposes static filters and the possibility to deactivate part of the instrumentation to avoid frequent events on later runs.

6. Conclusion and Future Work

We presented a methodology to perform binary instrumentation of OpenMP constructs. To this end, we integrated MAQAO with the scalable measurement system Score-P. We propose the SMINT and SMOMP interfaces. They can be used by other binary instrumentation tools and they can be extended to support other events and runtimes.

Our first experiments show that our approach is valid and has no additional overhead. The binary instrumentation does not need any recompilation, it does not interfere with the compiler optimization, and it provides additional implicit information for OpenMP such has the beginning and end of a reduction, or the implicit barriers.

The next steps will be to increase robustness and portability to ensure production quality. Furthermore we would like to explore the possibility of combining the results of the static performance analysis done by MAQAO with the runtime measurements. This would open up the opportunity of quantifying the actual performance degradation due to inefficiently implemented loops.

Acknowledgments

This work has been supported by the ITEA2 project H4H, the German Ministry of Education and Research, the French Ministry for Economy, Industry, and Employment, CEA, GENCI, Intel and UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

References

[1] Mpi forum : Mpi: A message-passing interface standard. version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, June 23rd 2009. Accessed: 2013-Jul-06.

[2] A. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, Liu X., E. Loh, and Lorenz D. Ompt: Openmp tools application programming interfaces for performance analysis. In *2013 International Workshop on OpenMP (IWOMP'13)*, Canberra, Australia, September 2013.

[3] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, Sameer S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*. Springer Berlin Heidelberg, 2012.

[4] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*. IOS Press, 2012.

[5] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Further improving the scalability of the Scalasca toolset. In *Proc. of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing, Reykjavik, Iceland, June 6–9 2010*. Springer, 2012.

[6] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008.

[7] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 2010.

[8] S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, May 2006.

[9] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*. Springer Berlin Heidelberg, 2010.

[10] H4h. `http://www.h4h-itea2.org/`, 2010. Accessed: 2013-Jul-08.

[11] M. Schlütter, P. Philippen, L. Morin, M. Geimer, and B. Mohr. Profiling Hybrid HMPP Applications with Sscore-P on Heterogeneous Hardware. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Munich, Germany, September 10-13, 2013*. IOS Press, 2013.

[12] Y. Kashnikov, P. Oliveira, E. Oseret, and W. Jalby. Evaluating architecture and compiler design through static loop analysis. In *IEEE Proceedings, , 2013 International Conference on High Performance Computing and Simulation (HPCS)*, 2013.

[13] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013.

[14] Andres S. CHARIF-RUBIAL. *On code performance analysis and optimisation for multicore architectures*. PhD thesis, October 2012.

[15] D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf. How to reconcile event-based performance analysis with tasking in openmp. In *Proc. of 6th Int. Workshop of OpenMP (IWOMP), Tsukuba, Japan*, Lecture Notes in Computer Science. Springer, June 2010.

[16] Karl Fürlinger and Michael Gerndt. ompp: A profiling tool for openmp. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008.

[17] Icc profile-guided optimization (pgo), 2011. Accessed: 2013-Jul-06.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[19] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[20] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 2000.

[21] J. Mussler, D. Lorenz, and F. Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11. Springer-Verlag, 2011.

[22] Nithammer C. Openmp validation suite. http://www.hlrs.de/organization/av/spmt/research/openmp-validation-suite/. Accessed: 2013-Jul-08.