A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P ⋆

Daniel Lorenz[1], Robert Dietrich[2], Ronny Tschüter[2], and Felix Wolf[1,3]

[1] German Research School for Simulation Sciences, 52062 Aachen, Germany
[2] Technische Universität Dresden, Center for Information Services and High Performance Computing, 01062 Dresden, Germany
[3] RWTH Aachen University, Department of Computer Science, 52056 Aachen, Germany

Abstract. The upcoming OpenMP tools interface (OMPT) has been designed as a portable interface for performance analysis tools. It provides access to OpenMP-related information at program runtime and can thus extend the analysis capabilities of current performance tools. This paper compares the functionality and convenience of OMPT with OPARI2 for event-based performance analysis. For this purpose, we integrated OMPT into the measurement infrastructure Score-P, which previously accessed OpenMP-related information using only source-level instrumentation with OPARI2. For comparison, we performed Score-P measurements of the NAS Parallel Benchmark suite and the LULESH code with OPARI2 instrumentation and with OMPT. In each case, we determined the overhead and evaluated the output. We found that the measurement overhead is dominated by the measurement system, while the contribution of the event source remains negligible. Moreover, OMPT and OPARI2 provide complementary views of the performance behavior. Whereas OPARI2 maintains a strictly source-code-centric perspective that reflects OpenMP standard abstractions, OMPT mirrors the behavior of the OpenMP runtime and exposes compiler optimizations.

1 Introduction

OpenMP is a widely used parallel programming specification for shared-memory platforms. Many compilers support it to exploit thread-level parallelism on modern hardware architectures. In the past, several analysis tools [5,6,8,9,11,16] that are capable of recording and displaying OpenMP related performance data emerged, assisting users in the optimization of their parallel programs. However, the OpenMP specification does not define a performance monitoring interface that enables tool developers to write portable measurement libraries. The emerging OpenMP tools interface (OMPT) [3] is intended to address this need.

In this work, we discuss OMPT in the context of the performance measurement infrastructure Score-P [9]. So far, Score-P captures OpenMP-related performance data using the source-to-source instrumenter OPARI2 [14]. However, since OMPT provides

⋆ This material is based upon work supported by the Department of Energy under Grant No. DE-FG02-13ER26158 / DE-SC0010668.

Published in "Using and Improving OpenMP for Devices, Tasks, and More", LNCS 8766, pp. 161-172, Springer, 2014. The original publication is available at www.springerlink.com

callbacks signaling the begin and the end of OpenMP constructs and other important events, it offers an attractive alternative to the current OPARI2-based instrumentation.

To evaluate the capabilities of OMPT for the event-based acquisition of OpenMP performance data, we integrated an OMPT adapter into Score-P and compared it in a set of different profiling and tracing scenarios with the default instrumentation via OPARI2. Our experiments are based on an OMPT implementation in the open-source version of the Intel OpenMP runtime [12]. To compare OMPT and OPARI2, we ran the NAS Parallel Benchmarks and the LULESH code, either with OMPT or with OPARI2 instrumentation. We determined the overhead for both cases, and evaluated the performance reports generated by Score-P.

The paper is organized as follows. Section 2 describes related work, in particular, further approaches to capture OpenMP-related performance data. Afterwards, in Section 3, we discuss the differences between the approaches taken by OPARI2 and OMPT in more detail. Section 4 outlines our prototypical OMPT adapter implementation in Score-P and the event model it adheres to. An evaluation of the experimental results is given in Section 5. Finally, in Section 6, we present our conclusions.

2 Related Work

Several tools [5,6,8,9,11,16] have emerged to provide insight into the parallel execution of OpenMP applications. Because there is no tools interface defined in the OpenMP specification yet, analysis tools rely on different methods to acquire their data. Performance tools based on sampling do not depend on an instrumenter like OPARI2. HPCToolkit [11], for example, collects the call path along with performance metrics at every sampling point. The routines are identified by their name and embedded in a calling-context. HPCToolkit recently implemented OMPT support to enable advanced analysis features like blame shifting for OpenMP applications. Another data acquisition strategy is instrumentation, which inserts hooks into the code to capture relevant events subject to further analysis. Advantages and drawbacks of the two approaches have been investigated in [13].

This work focuses on the measurement infrastructure Score-P [9], which uses instrumentation for data collection. Score-P is a joint performance measurement infrastructure for several analysis tools, including Vampir [8], Scalasca [6], TAU [16], and Periscope [1]. Currently, the performance-data collection of OpenMP events rests on the source-code instrumenter OPARI2 [14], which provides a portable way of instrumenting OpenMP pragmas by inserting calls to measurement functions around those pragmas into the program code. Score-P can produce both event traces and call-path profiles.

An alternative to OPARI2 is the ROSE compiler [15], which has been developed at Lawrence Livermore National Laboratory. It is an open-source compiler infrastructure to build source-to-source program translation and analysis tools for large-scale C/C++ and Fortran applications. ROSE can be used to identify and instrument all OpenMP 3.0 constructs in the source code [10]. Instead of transforming the source code, Paraver with Extrae as trace generator instruments OpenMP runtime routines based on a preloading mechanism (LD_PRELOAD) or DynInst [2]. The major drawback of this

approach is the limited portability, as currently only Intel, GNU and IBM OpenMP runtimes are supported. Furthermore, only visible OpenMP runtime library routines can be instrumented, which limits the obtainable information.

3 Instrumentation Approaches

Among event-based performance analysis tools, the instrumenter OPARI2 is widely used. OMPT also aims to support event-based tools. This section gives an overview of OPARI2 and OMPT in the context of event-based performance analysis.

3.1 OPARI2

OPARI2 [14] is a source-to-source instrumentation tool for OpenMP applications. It annotates OpenMP directives and runtime library calls with calls to the POMP2 measurement interface. A performance measurement infrastructure can implement these calls to obtain information about the execution of OpenMP parallel applications. OPARI2 is independent of a specific OpenMP implementation. It parses the source code and modifies it directly. To instrument a program using OPARI2, the application must be recompiled. The POMP2 event model provides events marking the begin and the end of an OpenMP construct. If an OpenMP construct refers to a structured block, OPARI2 inserts extra enter and exit events around this block—with the exception of loop constructs. If an OpenMP construct implies an implicit barrier, OPARI2 replaces the implicit barrier with an explicit barrier and instruments the explicit barrier. As an example, Listing 1.2 shows how OPARI2 instruments the OpenMP construct from Listing 1.1.

3.2 OMPT

OMPT [3] is an extension proposal for the OpenMP specification. It defines a standardized interface to obtain information from the OpenMP runtime system. OMPT pursues different objectives. First, the OpenMP tools API provides state information on the OpenMP runtime to be queried by an analysis tool. This feature is mainly intended for sampling. OMPT distinguishes three classes of states: mandatory, optional, and flexible. In contrast to optional states, mandatory states have to be maintained by all standard-compliant OpenMP implementations. Finally, implementations have some freedom if and when they indicate the transition to a flexible state.

Second, OMPT allows event-based performance tools to register function callbacks for events of interest. For most constructs, OMPT provides begin and end events that are triggered when a thread encounters a construct and when it finishes its execution, respectively. Furthermore, events exist to notify a tool when threads or tasks are created, a thread switches between the execution of two tasks, or a thread starts waiting or ends waiting in synchronization constructs. The event callbacks are classified as mandatory and optional. Mandatory events have to be implemented by all standard-compliant OpenMP implementations. The set of mandatory events is small but sufficient for basic performance analysis of OpenMP programs. For example, mandatory events include the start and the end of threads, tasks, and parallel regions. Nevertheless, the majority

Listing 1.1: Example of a simple OpenMP parallel loop

```
#pragma omp parallel for
for (i=0; i < 100000; i++)
    c[i] = a[i] + b[i];
```

Listing 1.2: Code generated by OPARI2 for a simple OpenMP parallel loop

```
POMP2_Parallel_fork( ... );
#pragma omp parallel  ...
{
    POMP2_Parallel_begin( ... );
    {
        POMP2_For_enter( ... );
        #pragma omp for nowait
        for (i=0; i < 100000; i++)
            c[i] = a[i] + b[i];
        {
            POMP2_Implicit_barrier_enter( ... );
            #pragma omp barrier
            POMP2_Implicit_barrier_exit( ... );
        }
        POMP2_For_exit( ... );
    }
    POMP2_Parallel_end( ... );
}
POMP2_Parallel_join( ... );
```

of information needed by Score-P is available only as optional events. An OpenMP implementation can support an arbitrary set of optional events and analysis tools can not rely on the availability of any optional event.

3.3 Comparison

Functionality and portability: OPARI2 accesses only the source code of an application. Therefore, OPARI2 is independent of a specific OpenMP runtime. The source code is parsed and rewritten by OPARI2 before the code is compiled which may alter code optimization decisions of the compiler. Performance measurement tools that want to use OMPT need an OpenMP runtime that implements OMPT. Neither error-prone source code parsing nor recompilation are needed with OMPT, shifting development and maintenance costs from tool to OpenMP runtime developers. It is only necessary to implement measurement adapters. However, tools have to live with the possibility that certain optional events are absent.

Obtainable information: The callbacks provided by OMPT and the instrumentation inserted by OPARI2 follow a similar event model. For example, both methods indicate

start and completion of a construct. On the other hand, the view the two methods provide of the application behavior is different in many regards. As a source-to-source instrumenter, OPARI2 has access to source-code information, but not to runtime information. Thus, it supplies many source-code details like the source-code location of constructs or additional clauses. Essentially, the instrumentation reflects the source-code structure and is agnostic of compiler optimizations. In contrast, OMPT is implemented in the OpenMP runtime library. Hence, it can access runtime information but lacks direct knowledge of the source code. It therefore does not know the original source-code structure but only the optimized binary code, which is why it can deliver insight into compiler optimizations. However, this implies that OMPT provides function pointer addresses for outlined functions of parallel regions and tasks as the only meta information on constructs. The function pointers can be used to obtain source code information if available. In principle, OMPT provides events for all constructs. However, most of the callbacks are optional in OMPT. Thus, the set of available events depends on the OpenMP implementation. With OPARI2, all events are always available, but a user can disable the instrumentation of any set of constructs. Additionally, OMPT allows direct measurement of waiting time in synchronization constructs. With OPARI2 a user can only assume waiting time if the execution time of a synchronization construct is large. Figure 1 shows the respective event trace of the OpenMP parallel loop construct from Listing 1.1 in the Vampir trace browser. The upper two charts (white background) depict the event trace recorded with OMPT callbacks. A timeline representation of the parallel loop execution with four threads is illustrated in the first chart and the corresponding call stack of the master thread is shown in the second chart. The lower two charts (purple background) present the data obtained from the OPARI2 instrumentation for the execution of the same OpenMP construct. OMPT reflects that the OpenMP runtime completes the execution of the parallel loop with an implicit barrier (blue region), whereas the OPARI2 instrumentation inserts an explicit barrier to the parallel loop construct.

4 Score-P Implementation

For the purpose of this study, we implemented a Score-P prototype supporting the OMPT interface. It was our goal to measure OpenMP applications even if the OpenMP runtime implements only mandatory events. If optional events are available, they should enrich the measurement with additional information. The Score-P architecture [9] consists of an adapter layer which captures events and a measurement layer which passes the data to the profiling or tracing backend. Event traces are written in the Open Trace Format 2 (OTF2) [4], which can be analyzed with Vampir or Scalasca. Call-path profiles are stored in the CUBE4 format.

First, we implemented a new support component for the internal thread management under OMPT, making no assumption about the availability of optional events. Unfortunately, the callbacks indicating the begin and end of an implicit task are optional events in OMPT. However, the information when a worker thread starts or ends its execution is essential in Score-P. Although we can estimate these times from the begin and end of a parallel region on the master thread, we still need to know which threads belong

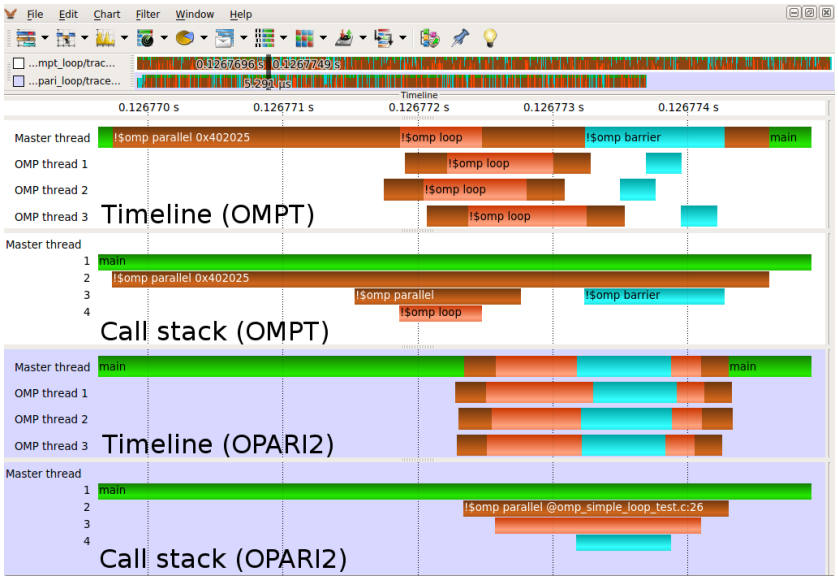


Fig. 1: OMPT (white background) and OPARI2 (purple background) perspective on an *OpenMP parallel for* region executed with four threads. The Vampir compare view shows the timeline and call stack view on the respective event traces.

to the parallel region. However, Score-P only learns that a thread is executing a parallel region if this thread triggers an event inside the region, which is not guaranteed. Thus, if no events appear inside a parallel region, Score-P can show that there is a parallel region but does not know about any worker threads running inside.

Second, we developed a new adapter that implements the OMPT callback functions and translates the call-backs to Score-P events. The OpenMP runtime version that we used for our experiments does not implement all optional events. However, for most OpenMP constructs that occurred in our experiments, OMPT call-backs exist.

5 Evaluation

In this section, we compare OPARI2 and OMPT on the basis of performance experiments with several benchmarks. We ran the NAS Parallel Benchmark (NBP) suite in profiling mode, while we ran LULESH in tracing mode. For each test case, we chose the minimum execution time of ten measurements.

5.1 Profiling Overhead

Our test platform was the the Linux cluster JUROPA at Forschungszentrum Jülich. JUROPA has 2208 compute nodes, each equipped with two Intel Xeon X5570 quad-core processors running with 2.93 GHz We used the Intel compiler version 11.1 to build the

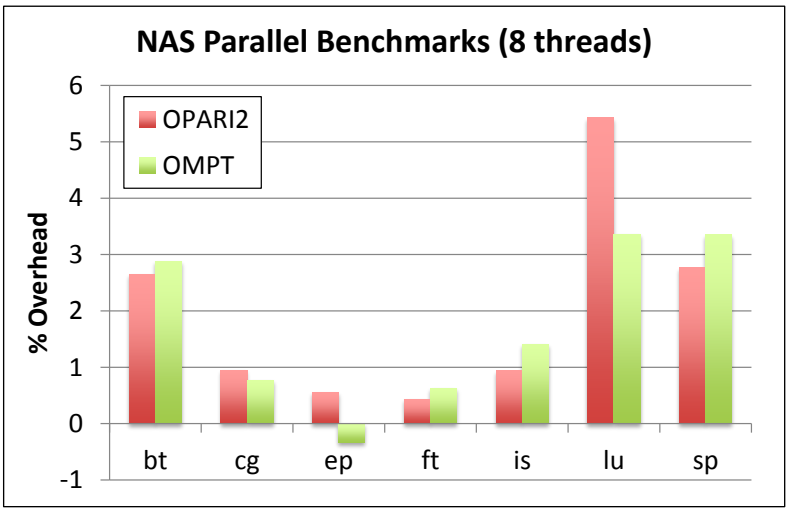


Fig. 2: Overhead of NPB measurements with OPARI2 instrumentation and with OMPT callbacks on JUROPA.

OpenMP runtime, Score-P and the NPB suite. The number of threads was always eight. We measured the runtime of (i) the uninstrumented codes, (ii) with OPARI2 instrumentation of OpenMP constructs, and (iii) with OMPT callbacks to record OpenMP related data. In addition to OPARI2 instrumentation and OMPT callbacks, we instrumented the main function manually. The measured overheads are shown in Figure 2.

The overhead is low in all cases. Only the OPARI2-instrumented `lu` benchmark shows an overhead of 5.4%. In all other cases the overhead is less than 3.4%. For `cg`, `ep`, and `ft`, the overhead is even less than the standard measurement deviation for these applications. With the exception of `lu`, the overheads of the OMPT and OPARI2 instrumentation are very similar. The difference in runtime is less than the standard deviation. The negative overhead measured for the OMPT instrumented `ep` is due to measurement deviation.

Only in the `lu` benchmark code, we observed a significant difference between the overheads of OMPT call backs and OPARI2 instrumentation. The reason is due to the more than 140,000,000 visits to OpenMP `flush` constructs in `lu`. They are instrumented by OPARI2, where they produce more than 95% of the events, but do not trigger any call backs in our OMPT implementation. This is because the Intel OpenMP runtime has no support for flush callbacks yet.

Except for the creation of a new system thread, the Score-P measurement system performs no communication or synchronization between threads during the measurement. Thus, we expect Score-P to be embarrassingly parallel. Measurements with the `lu` benchmark on JUROPA with up to 16 threads show that the speedup of the OPARI2 instrumented code, the OMPT based measurement and the uninstrumented code are identical.

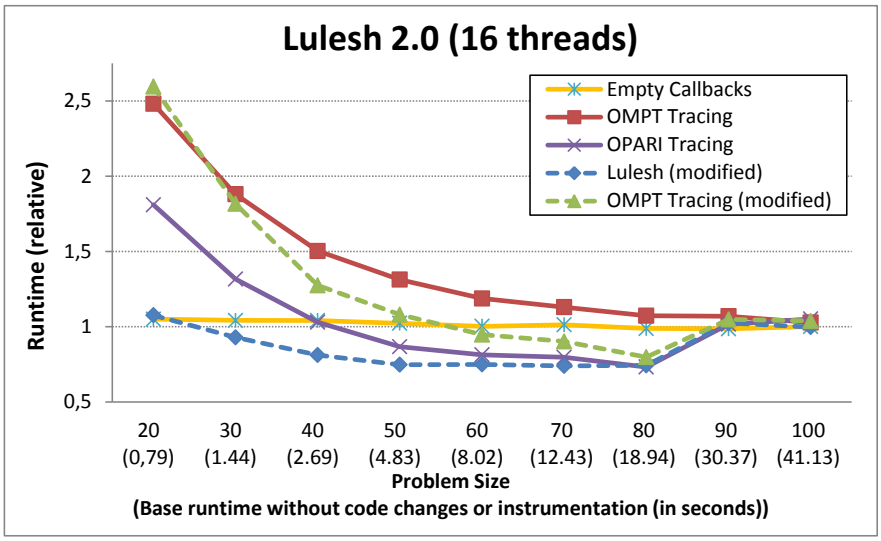


Fig. 3: Runtimes of LULESH with OPARI2 instrumentation and OMPT callbacks. Runtimes are relative to the runtime of the uninstrumented LULESH code. In the modified (and also uninstrumented) version, the implicit barrier was manually exchanged for an explicit barrier, similar to the transformation performed by OPARI2 (Listings 1.1 and 1.2).

5.2 Tracing Overhead

LULESH [7] is a shock hydrodynamics code developed at Lawrence Livermore National Laboratory. It is known to challenge machine performance. Furthermore, it stresses compiler vectorization, OpenMP overheads, and intra-node parallelism. For the latter, it employs for loops in simple non-nested parallel regions. We used LULESH version 2.0 and conducted our experiments on the Sandy Bridge partition of the HPC cluster Taurus at Technische Universität Dresden. We ran the job with 16 threads on a single node, which is equipped with two Intel Xeon CPU E5-2690 (8 cores) at 2.90GHz and hyperthreading disabled. LULESH, the open source version of the Intel OpenMP runtime, and the measurement system Score-P were compiled with the Intel compiler version 13.0.1. To record a similar set of events with both instrumentation approaches and enable the visualization of worker threads, we inserted calls to the implicit task begin and end callbacks into the the Intel OpenMP runtime.

We ran LULESH with different problem sizes for a fixed number of 200 iterations. Figure 3 shows the the runtimes relative to the original LULESH code without instrumentation. The runtime of the LULESH code without any instrumentation increases from $0.79sec$ for a problem size of 20 to $41.13sec$ for the problem size 100. The problem size defines the workload and increases the total computation time, whereas the number of OpenMP events stays constant. As the number of measurement events is independent of the problem size, large instrumentation overheads can be forced with small problem sizes and vice versa. For problem sizes smaller than 90, the OPARI2

Construct	bt		ft	
	OPARI2	OMPT	OPARI2	OMPT
atomic	2	–	0,	–
barrier	19	aggregated	9,	aggregated
loop	30	aggregated	8	aggregated
master	5	aggregated	1	aggregated
parallel	10	10	9	10

Table 1: Number of constructs distinguished by OPARI2 and OMPT for `bt` and `ft`. The OMPT implementation in our version of the Intel OpenMP runtime does not yet support atomic regions. For most OpenMP constructs, OMPT-based measurements merge constructs of the same type and aggregate their data.

instrumented version has between 9 and 73 percentage points less overhead than the OMPT version. For problem sizes from 50 to 80, the OPARI2 instrumented code was even faster than the uninstrumented code. During instrumentation, OPARI2 substitutes explicit barriers for implicit barriers, as depicted in Section 3.1. If we apply this change manually to the LULESH source code, the uninstrumented code runs up to 26% faster than the unmodified version. As a consequence, the measurements with OMPT are much faster, too. The measurements with the modified code are shown in Figure 3 as dashed lines. We believe that this change alters the compile-time optimization decisions and constitutes the major reason why the OPARI2-based and the OMPT-based measurements are different.

5.3 Structural Differences in Performance Content

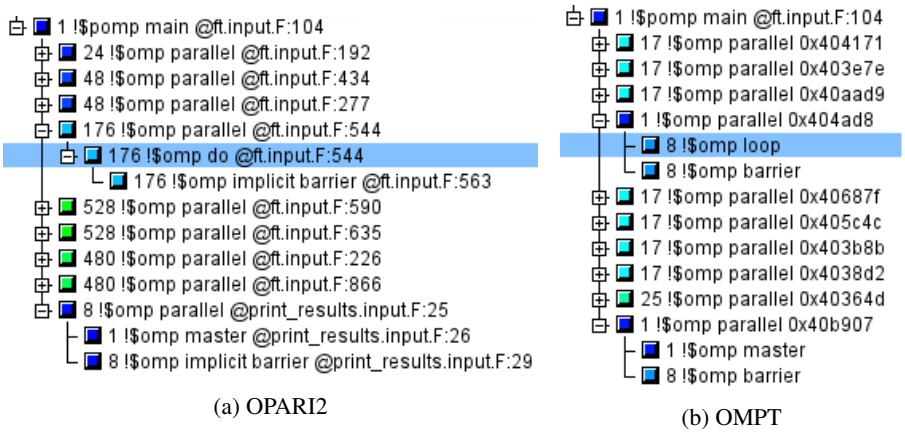
In the following, we highlight structural differences in the output of the two methods. As these differences are based on the instrumentation technique, our observations apply to both profiling and tracing equally.

The first observation is that OPARI2 provides source code information on every OpenMP construct, which allows the user to distinguish them during analysis. Except for parallel constructs and tasks, OMPT does not provide any information to distinguish constructs of the same type. Thus, for all remaining construct types, constructs of the same type appear merged and their performance data aggregated. This can be illustrated with profiling data from NPB. Since OPARI2 measurements can distinguish multiple OpenMP constructs of the same type appearing inside the same call path, the call tree of the OPARI2-instrumented code might look more differentiated. Table 1 shows the number of distinguishable constructs OPARI2 and OMPT recognize. To quantify the additional information the distinction among constructs of the same type provides, we counted the number of call paths (i.e., nodes) in the call tree (Table 2).

Table 2 shows that in most cases the OPARI2 profiles contain significantly more call paths than the OMPT profiles. For `bt`, `cg`, and `lu`, the OPARI2 profiles show more than twice as many call paths as the OMPT profiles. A remarkable exception is `ft`, where the OMPT instrumentation leads to more call paths than the OPARI2 instrumentation

		bt	cg	ep	ft	is	lu	sp
OPARI2	Visits	47,994	147,990	154	2682	810	148,517,435	133,690
	Call paths	67	54	12	28	23	95	79
OMPT	Visits	34,197	127,571	54	172	626	1,712,033	90,265
	Call paths	32	26	11	31	18	34	44

Table 2: Number of visits and different call paths in the profile of NPB codes, measured with OPARI2 and OMPT.



(a) OPARI2

(b) OMPT

Fig. 4: Comparison of the `ft` call trees generated with OPARI2 and with OMPT.

because it shows one additional outlined function for a parallel construct. Both profiles are shown in Figure 4a and Figure 4b. The source code contains nine parallel constructs, as identified by OPARI2.

A second observation are the different visit counts produced by the two methods. This has several reasons. First, our OMPT implementation did not yet support all constructs that appeared. For example, in `lu` more than 95% of the visits in the OPARI2 instrumented measurement stem from flush constructs, for which the OMPT implementation does not produce events yet. Another reason is that the OPARI2 module of Score-P counts the start and the end of the implicit task inside the parallel region for every thread. Because the implicit task begin event is optional in OMPT, the OMPT module in Score-P must rely on the begin of the parallel region itself, which happens only on the master thread. However, this could be easily changed in a post-processing step, or by implementing the implicit task-begin event callback in the OpenMP runtime.

In some cases, compiler optimizations affect the visit count, which is the number of times a call path has been visited. For example, a loop in the main routine of `ft` iterates over subroutine calls to `evolve`, `fft`, and `checksum`, which contain parallel constructs. The OPARI2 measurement shows 20 or more visits per thread for these parallel constructs. However, the OMPT measurement shows only one visit for each of its 10 parallel constructs. Our explanation is that the compiler applied optimizations, e.g.,

moved the parallel region creation around the loop. Optimizations like unrolling are also a possible explanation why the number of parallel constructs in the OMPT result for `ft` differs from the number of parallel constructs in the source code. Furthermore, optimizations may result in outlined functions which cannot be easily mapped to the user code. We recompiled `ft` with optimization level zero to prove this assumption. The measurement result of the unoptimized `ft` with OMPT shows 9 outlined functions for parallel constructs. However, every parallel construct is still visited only once.

Obviously, OMPT measurements may provide insight into compiler optimizations. However, understanding this information may require knowledge of the compiler and may even then not be comprehensible at the first glance. In contrast, OPARI2 delivers information that strictly reflects the source code.

6 Conclusion

We compared OMPT callbacks and OPARI2 instrumentation with respect to their suitability for event-based performance measurements. OPARI2's source-to-source translation approach neither has access to object code nor to intermediate representations, but reflects the structure of the source code very well. An advantage is that a user can easily map the measurement results onto his mental image of the program. Since OPARI2 passes along all relevant source code information, the results can even be explored in a source-code browser. On the other hand, the instrumentation may interfere with compiler instrumentation and optimization.

For the development of OMPT, one of the initial design guidelines was to create an interface that can be implemented in the OpenMP runtime without having to change the compiler. Thus, it provides a view of the OpenMP runtime level, including compiler optimization artifacts. This may reflect the execution behavior of the application more accurately. On the other hand, differences to the source code representation may obscure measurement results sometimes. Overall, OPARI2 and OMPT provide complementary information, which makes it reasonable to combine both approaches. The information gathered with OMPT could be extended with source code correlation via OPARI2, whereas events that are not available with OPARI2 (e.g. thread begin/end, wait barrier begin/end) could be captured using OMPT. The measurement overhead is generally low for both, OMPT and OPARI2. In most cases, the measurement system itself dominates the overhead regardless of the instrumentation method. However, in cases where the source code instrumentation of OPARI2 interferes with compiler optimization, OPARI2 instrumentation may lead to different execution times.

The mandatory set of callback functions in OMPT allows call-path profiles to be constructed from the events produced by our instrumentation—provided that no tasks are used. However, if a worker thread does not trigger any events inside a parallel region, it may remain invisible in the measurement. To construct call-path profiles for tasks, the measurement system must be notified of task switches. Thus, to support event-based performance tools, we recommend to support at least the optional callbacks for the events `ompt_event_implicit_task_begin` and `ompt_event_task_switch` in OMPT implementations.

References

1. S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, Berlin/Heidelberg, 2010.
2. Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, November 2000.
3. Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT: An OpenMP tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2013.
4. Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix 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*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.
5. Karl Fürlinger and Michael Gerndt. ompp: A profiling tool for openmp. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008.
6. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Daniel Becker Erika Abraham, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.
7. Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.
8. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 139–155, Stuttgart, Germany, July 2008. Springer-Verlag.
9. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*, pages 79–91. Springer Berlin Heidelberg, September 2012.
10. Chunhua Liao, Daniel J Quinlan, Thomas Panas, and Bronis R de Supinski. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 15–28. Springer, 2010.
11. Xu Liu, John Mellor-Crummey, and Michael Fagan. A new approach for performance analysis of OpenMP programs. In *Proceedings of the 27th international ACM conference on supercomputing*, pages 69–80. ACM, 2013.
12. John Mellor-Crummey et al. OMPT support branch of the open source Intel OpenMP runtime library. `http://intel-openmp-rtl.googlecode.com/svn/branches/ompt-support`, December 2013.
13. Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance sata collection using a hybrid approach. *SIGSOFT Software Engineering Notes*, 30(5):126–135, September 2005.

14. B. Mohr, A.D. Malony, S.S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.
15. D. J. Quinlan et al. ROSE compiler project. http://www.rosecompiler.org, April 2014.
16. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.