Improving Tool Support for Nested Parallel Regions with Introspection Consistency

Vladimir Indic[1](✉) and John Mellor-Crummey[2]

[1] Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
vladaindjic@uns.ac.rs

[2] Department of Computer Science, Rice University, Houston, TX 77251-1892, USA
johnmc@rice.edu

Abstract. The OpenMP 5 standard defines OMPT—an application programming interface for tools that includes a set of introspection routines. At any point in time, a sampling-based performance tool may invoke these introspection routines from a signal handler to inquire about the nesting of parallel and task regions. Unfortunately, the OpenMP 5 standard doesn't precisely specify what one may observe with these routines when monitoring a program as it executes nested parallel regions. To address this shortcoming, we propose that the OpenMP standard require that an OpenMP implementation supports *introspection consistency*. This paper defines introspection consistency, describes why tools need it, and explains a novel strategy for implementing it using wait-free coordination between an OpenMP implementation and its OMPT introspection routines. We describe an implementation of this technique in the LLVM OpenMP runtime and evaluate the runtime overhead of supporting introspection consistency in LLVM OpenMP using a microbenchmark for nested parallel regions and SPEC OMP2012.

Keywords: OpenMP · OMPT · wait-free coordination

1 Introduction

The OpenMP Application Programming Interface (API) defines a directive-based programming model for harnessing parallelism within nodes that employ one or more multicore processors and sometimes accelerators. Since the inception of OpenMP, providing tools that support two or more OpenMP implementations has been a challenge. The substantial semantic gap between an OpenMP program and its implementation, compounded by differences between OpenMP implementations, makes it difficult for tools to attribute performance metrics

This work was supported in part by Faculty of Technical Sciences, Novi Sad, Serbia and the Exascale Computing Project (17-SC-20-SC)—a collaborative effort of the U.S. DOE Office of Science and the National Nuclear Security Administration.

to a source-level calling contexts. To bridge this gap, Eichenberger et al. [5] proposed OMPT—an API for performance analysis and correctness tools. In 2018, OMPT became an integral part of the OpenMP 5.0 standard [10].

The OMPT API defines a set of runtime entry points designed to support asynchronous introspection of the state of an executing OpenMP program by a sampling-based performance tool. To retrieve information about the current parallel (respectively, task) region, a sampling-based performance tool asynchronously invokes the `ompt_get_parallel_info` (`ompt_get_task_info`) runtime entry point passing `ancestor_level` = 0; information about enclosing parallel (task) regions can be obtained by specifying values of `ancestor_level` > 0. To date, not enough attention has been paid to the semantics of these routines. The standard simply states that `ompt_get_parallel_info` (respectively, `ompt_get_task_info`) returns

- 2 if a parallel (task) region exists at the specified ancestor level and information about the parallel (task) region is available,
- 1 if a parallel (task) region exists at the specified ancestor level but information is currently unavailable, and
- 0 otherwise.

While integrating support for the OMPT interface into Rice University's HPCToolkit performance tools [1,13], which use asynchronous sampling to collect call path profiles and traces [12] for CPU threads as a program executes, it became clear that this definition of the semantics for the `ompt_get_parallel_info` (`ompt_get_task_info`) introspection routines is too weak. There are several problems.

- After an OpenMP program (1) enters a parallel (respectively, task) region and (2) the OpenMP implementation provides information about that region to a tool by returning 2 to an introspection query, there is no requirement that the OpenMP introspection routine must continue to return information about the region until the program begins to exit the region.
- If a tool stores into the `ompt_data_t parallel_data` (respectively, `ompt_task_data`) word maintained by an OpenMP implementation for a parallel (task) region, a tool will not be reliably able to use an OpenMP introspection routine to retrieve this data throughout the lifetime of the region.

These problems are not just theoretical. We have observed both issues in practice while using HPCToolkit to profile an OpenMP program linked againts the LLVM OpenMP runtime. When a thread receives a sample, it executes HPCToolkit's signal handler code and unwinds the call stack to determine the context that incurs cost. Worker threads of OpenMP parallel regions can only determine partial call paths because they are unaware of the region's invocation context known only to the primary thread. To compute a full, user-level calling context, a worker thread subscribes to receive the region context from the primary thread by updating a tool data structure whose pointer is stored in the active region's `parallel_data` word. At the end of the parallel region, the primary thread reads `parallel_data` to obtain the pointer to the tool data structure, unwinds the call stack to determine the region's context, and shares it with subscribed worker threads that then asynchronously assemble full call paths for samples received while executing the region.

In the LLVM OpenMP runtime, we noticed timing windows while creating nested parallel regions during which HPCToolkit receives wrong information about enclosing parallel regions, causing a sampled thread to be unable to determine if it is a worker thread that should subscribe to receive the region's context. Furthermore, runtime might override the content of an enclosing region's `parallel_data` while creating a nested parallel region; this causes the loss of the worker threads' subscriptions, which causes the profiler to fail.

Such issues can be avoided by improving the OpenMP specification and tightly integrating the implementation of OpenMP parallel and task regions with their OMPT introspection routines. This paper makes the following contributions:

- We propose that the OpenMP specification requires that an OpenMP implementation supports *introspection consistency*; in brief, this requires that when a program is executing in an OpenMP parallel or task region, the region must be visible to OMPT introspection routines.
- We explain why introspection consistency is needed to support reliable sampling-based performance tools for call path profiling.
- We describe a novel approach for implementing introspection consistency using wait-free coordination between an OpenMP implementation and its OMPT introspection routines.
- We overview an implementation of introspection consistency in LLVM OpenMP.
- We evaluate the runtime overhead of supporting introspection consistency in LLVM OpenMP using a microbenchmark for nested parallel regions and SPEC OMP2012.

Section 2 briefly describes the implementation of parallel regions, tasks[1], and OpenMP introspection routines in LLVM OpenMP to provide context for understanding our contributions. The remaining sections motivate introspection consistency, describe a high-level approach to providing it, describe our implementation in LLVM OpenMP, evaluate its cost, discuss related work, and present our conclusions.

2 Background

This section briefly describes the implementation of parallel regions, serialized parallel regions, implicit and explicit tasks, and OMPT introspection support for nested parallel and task regions in LLVM OpenMP runtime.

Parallel Regions and Implicit Tasks. When a thread encounters a parallel region construct that at least two threads should execute, it must form a team of threads

[1] We use the word "task" as a synonym for a task region for brevity.

that will execute that region. The encountering thread becomes the primary thread of the region's team. Prior to entering this region, the primary thread initializes the team descriptor, records the number of threads in the team, sets the descriptor's parent pointer to the enclosing parallel region (if any), assembles a team of threads for the region, shares the descriptor with the team of threads, and sets its own current team descriptor to the region's team descriptor. Then, each thread of the region's team begins an implicit task that executes the code for the region. Before scheduling the implicit task for execution, a thread assembles the task descriptor, updates its team and scheduling parent pointers to link the parallel region's team descriptor and the enclosing task, respectively, and finally sets its own current task descriptor to the assembled task descriptor. After every thread completes the work of its implicit task and synchronizes with a final barrier, it destroys the current implicit task descriptor. Afterward, the primary thread destroys the parallel region's team descriptor and resumes work in the context of the region's parent.

Explicit Tasks. An explicit task may suspend its execution after the creation. Furthermore, multiple threads may schedule and suspend the execution of an untied explicit task, causing the runtime to update the task's scheduling parent pointer in its task descriptor when the untied task was rescheduled.

Serialized Parallel Regions. A serialized parallel region is executed by a single thread. Serialized parallel regions occur frequently enough that OpenMP implementations often provide a tailored implementation for high performance. Here, we discuss the implementation of nested serialized parallel regions in the LLVM OpenMP runtime.

When a thread executing a serialized parallel region R_1 encounters a parallel region construct that yields another serialized region R_2, it reuses the current team descriptor associated with R_1 for R_2 rather than allocating and initializing a separate descriptor for R_2. Similarly, the thread reuses the current task descriptor associated with the R_1's implicit task to represent the R_2's implicit task. This approach is roughly 80% faster than using separate descriptors; however, it leads to problematic behaviors for both `ompt_get_parallel_info` and `ompt_get_task_info`.

The lack of separate team and task descriptors leads to losing important OMPT information about nested serialized parallel regions and corresponding implicit tasks. To overcome this problem, LLVM runtime developers introduced a separate lightweight team descriptor (lwt) to store OMPT information about nested serialized parallel regions and associated implicit tasks. When creating nested serialized parallel region R_2, the current team and task descriptors contain OMPT information about R_1 and its implicit task. The thread allocates a new lwt descriptor, fills it with OMPT information associated with R_1 and the corresponding task from the current team and task descriptors, and then overwrites the OMPT information in those descriptors with information about R_2 and its implicit task. The lwt descriptor for R_1 is then pushed into a linked list, known hereafter as the lwt list. After executing R_2's implicit task, the process is reversed. The thread removes R_1's lwt descriptor from the lwt list and copies OMPT information about R_1 and its implicit task back to the current team and task descriptors, overwriting the OMPT information about region R_2 and the corresponding task which recently completed.

Tool Support for Parallel and Task Regions. For each `ancestor_level` level at which information is available, `ompt_get_parallel_info` will return the team size for the parallel region and a pointer to a `parallel_data` word provided for the region by an OpenMP implementation for use by a tool. Similarly, a tool might inspect the OMPT state of an active task region by invoking `ompt_get_task_info`. This routine reveals the type of the task, procedure frame information for that task, and the number of the thread in the parallel region executing the task. `ompt_get_task_info` provides the `task_data` word associated with the task and maintained by the runtime for use by a tool as well as `parallel_data` word for the region that contains the task.

While a thread executes a parallel region, `ompt_get_parallel_info` reads OMPT information from the current team descriptor and either follows the parent pointers of region team descriptors or a chain of lightweight team descriptors that contain information about enclosing serialized parallel regions. The routine `ompt_get_task_info` is similar, returning information about the nesting of explicit tasks, implicit tasks, and their associated parallel regions. It reads the information from the current task and team descriptor and eventually follows the chain of task and team (lwt) descriptors in pairs.

3 Approach

As described in the Introduction, having tool data associated with parallel and/or task regions be lost or unavailable as a program executes is unacceptable for tools. To address these issues, we propose introspection consistency to avoid having tool data become unavailable while a parallel region or task is active.

An OpenMP implementation provides *introspection consistency* if it obeys the following principles:

- A thread that is part of a team for a parallel region must provide information about the region and its implicit task to the OMPT introspection routines `ompt_get_parallel_info` and/or `ompt_get_task_info` upon request from the time of its *implicit-task-begin* event until the *implicit-task-end* event of the primary thread in the region.
- A thread must provide information about a tied explicit task to the OMPT introspection routine `ompt_get_task_info` upon request from the time the task is scheduled on a thread until the task completes.
- A thread must provide information about an untied explicit task to the OMPT introspection routine `ompt_get_task_info` upon request from the time the task is scheduled for execution on a thread until it suspends.

When creating a parallel region, one must associate a thread with descriptors for both a team and an implicit task. One approach for entering a parallel region is to atomically push a (team, task) pair of descriptors representing a parallel region and its implicit task. However, that approach will require that the runtime always accesses the descriptors with an additional level of indirection. The LLVM OpenMP runtime maintains the current task and team descriptors separately to avoid this extra level of indirection. When creating a new parallel region, the runtime first updates the current team descriptor to the new region and then the current task descriptor to the region's implicit task. With this approach, one must be careful, or `ompt_get_task_info` might, for instance, read OMPT information from the current team descriptor matching the new innermost parallel region and the current task descriptor corresponding to an implicit task of an enclosing parallel region (if any). In the upstream LLVM OpenMP, this causes inconsistent results from `ompt_get_task_info`.

To avoid such inconsistencies, in our improved implementation, `ompt_get_task_info` first reads OMPT information from the current task descriptor and then accesses the current team descriptor. Each task descriptor links the descriptor of the team to which it belongs. Suppose `ompt_get_task_info` finds that the current task descriptor does not reference the current team descriptor, meaning that the current task descriptor corresponds to the implicit task of an enclosing parallel region. In that case, `ompt_get_task_info` will report the presence of the inner parallel region but indicate that it cannot provide information about the region. `ompt_get_task_info` will not provide information about a parallel region and its implicit task until the runtime updates both the team and task descriptors for the region and its task. A tool can still access the information about the inner parallel region by invoking `ompt_get_parallel_info`, and thus detect the creation/destruction of the region.

We encountered a different obstacle to providing introspection consistency for nested serialized parallel regions. As described in Sect. 2, the implementation of nested serialized parallel regions is optimized to avoid allocation and full initialization of a descriptors for nested serialized parallel regions and corresponding implicit tasks. However, the runtime fails to preserve introspection consistency for an enclosing serialized parallel region R_1 and its implicit task during the creation (and respectively, destruction) of a nested serialized parallel region R_2. The signal handler might interrupt the runtime during R_2's creation and store a value for `parallel_data` for R_1 (for `task_data` for R_1's implicit task) after the runtime has captured `parallel_data` for R_1 (`task_data` for R_1's implicit task) to move it into a lightweight team (lwt) descriptor, causing a value of `parallel_data` (`task_data`) to be lost.

The runtime can trivially overcome this problem by blocking all signals during the creation/destruction of a nested serialized parallel region to prevent signal delivery and tool introspection while information about a serialized parallel region and its implicit task is being updated. However, in the case of frequent short nested serialized parallel regions, blocking and unblocking signals can significantly interfere with sampling and cause a tool to collect unrepresentative data. Instead, we have the runtime and an introspection routine invoked from a signal handler employ wait-free coordination [7] to achieve consensus about the information associated with R_1 and R_2. The next section describes a protocol that enables a call to `ompt_get_parallel_info` or `ompt_get_task_info` from a signal handler to recognize when the runtime is in the process of moving data to prepare for a nested parallel region. In that case, the introspection routine finishes preparing the nested region so it can return information about both the nested and enclosing regions. It atomically writes this information into the runtime's current team, task, and lwt descriptors to ensure that the runtime will observe it after the introspection routine finishes.

4 Implementation

In this section, we describe the implementation of a novel wait-free protocol for coordinating the LLVM OpenMP runtime with its OMPT introspection routines to maintain introspection consistency for nested serialized parallel regions. For this purpose, we extended each of the team, task, and lwt descriptors with a pair of OMPT descriptors (depicted and numbered with 1 and 2 in Fig. 1) and a pointer (shown as `ptr` in Fig. 1) that indicates which descriptor of the pair contains valid OMPT information about the OpenMP parallel/task construct. The runtime and an introspection routine called from a tool's signal handler use wait-free coordination to achieve consensus about which OMPT descriptor associated with a nested serialized parallel region (and its implicit tasks) contains valid state by atomically updating the value of the `ptr` pointer to reference one of the pair's descriptors.

Figure 1a depicts the descriptors containing information associated with two serialized parallel regions, R1 and R2, and their corresponding implicit tasks when the primary thread executes the nested region R2. As the `ptr` pointer (shown in blue) of the heap-allocated team descriptor indicates, the R2's OMPT information resides in the first OMPT descriptor of the pair. The `ptr` (shown in red) of the task descriptor allocated on heap indicates that the second OMPT descriptor of the corresponding pair contains the information associated with R2's implicit task. The first OMPT descriptor of the pair belonging to the stack-allocated lwt descriptor depicted in the right half of Fig. 1a contains information about the enclosing parallel region R1 and its implicit task. This lwt descriptor is the only element of the lwt list at the moment, meaning that the list's head pointer (`lwt_list`) references this descriptor.

When a primary thread encounters a parallel construct yielding another serialized region R3 while executing region R2, it starts executing runtime code responsible for assembling a new serialized parallel region. The thread allocates a new lwt descriptor (bottommost in the Fig. 1b) for storing the OMPT information associated with region R2 and its implicit task. As Fig. 1b depicts, the thread sets the lwt's `ptr` to 0, meaning that consensus is not achieved yet. Similarly, this thread marks the `ptr` pointers of the current team and task descriptors by setting their least significant bits to 1 to announce that the information about

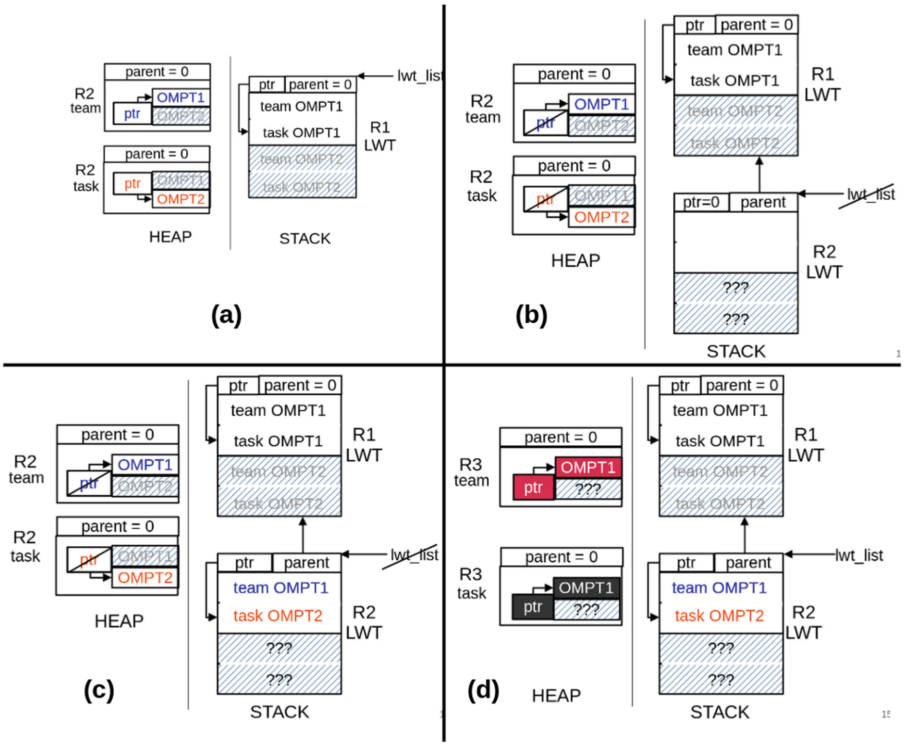

Fig. 1. Migrating data from the current team and task descriptors to a newly allocated lwt descriptor while creating a nested serialized region. OMPT descriptors belonging to the current team, task and lwt descriptors are numbered with the corresponding index in the pair (1 or 2). A slash over a pointer indicates the pointer has been marked by setting its least significant bit to 1. Shaded fields do not contain information of interest. (Color figure online)

the R2 region and its implicit task will be moved shortly after. Suppose a tool receives a sample at this moment. In that case, the signal handler invokes the introspection routine to inspect the information about R2 and its implicit task. The routine masks team and task `ptr` pointers by removing marks and reads the contents of the addresses referenced by masked pointers.

Afterward, the primary thread sets the new lwt's parent pointer to reference the `lwt_list` head pointer, marks the address of the lwt, and updates the `lwt_list` pointer with the marked address. As a result, the lwt is inserted at the end of the lwt list. The updated marked `lwt_list` head pointer indicates that the process of copying OMPT information requiring wait-free coordination is in progress, meaning that the introspection routine is responsible for finishing it if called from the signal handler.

As Fig. 1c depicts, the primary thread copies the OMPT information associated with region R2 and its implicit task from the locations indicated by the `ptr` pointers of the team and task descriptors to the first OMPT descriptor of the pair belonging to the recently allocated lwt. Subsequently, the thread tries to atomically update the value of the lwt's `ptr` with the address of the previously updated OMPT descriptor by using compare-and-swap. As shown in Fig. 1c, the primary thread decides that the first OMPT descriptor of the pair contains valid information associated with R2 and its implicit task. Otherwise, suppose a tool receives a sample before the primary thread updates the lwt's `ptr`. In that case, the introspection routine invoked by the tool's signal handler and executed by the same primary thread decides that the second OMPT descriptor of lwt's pair contains the valid information, and the compare-and-swap executed by the runtime fails.

Afterward, the primary thread tries to reuse the current team and task descriptors depicted on the left in Fig. 1d for storing the information associated with the new R3 region and corresponding implicit task. To do so, it initializes the content of the first OMPT descriptors of team and task descriptor pairs (solid red and dark grey, respectively) to store the desired OMPT information. Subsequently, it tries to update the values of the corresponding `ptr` pointers to reference the initialized OMPT descriptors using compare-and-swap. As in the case of copying the OMPT information about the R2 and its task to the lwt descriptor, the update succeeds if no call to the introspection routine by a tool's signal handler interrupts the runtime execution. Otherwise, the first call to an introspection routine finishes the update by deciding that the second OMPT descriptor of the team (task) descriptor's pair contains the valid information about the region R2 (R2's implicit task). Finally, the primary thread clears the mark bit of `lwt_list`, indicating that the preparation of the new serialized parallel region R3 is complete.

If an introspection routine sees that `lwt_list` is marked, it recognizes that the runtime is in the process of updating and initializing OMPT information about the two innermost serialized parallel regions (and associated tasks). The introspection routine finishes the update. Since the primary thread executes the introspection routine atomically with respect to the runtime code, the introspection routine need not use compare-and-swap to update the `ptr` pointers shared with the runtime. The introspection routine examines whether the lwt's `ptr` is equal 0 to observe if the runtime finished moving OMPT information from the current team and task descriptors to the lwt. Similarly, it examines if the `ptr` pointers of team and task descriptors are marked, meaning the runtime still has not reused the team and task descriptors for storing OMPT information about the innermost parallel region and its task. The introspection routine uses the second OMPT descriptors of the pairs that belong to the team, task, and lwt descriptors while moving and initializing OMPT information associated with the innermost serialized regions. After finishing the process of migrating the OMPT information for these regions, the introspection routine provides OMPT information about all active parallel regions and tasks to the tool.

5 Evaluation

This section evaluates four standard-compliant versions of the LLVM OpenMP runtime. The versions differ in how they implement nested serialized parallel regions and support OMPT introspection. Consider how they handle a program executing serialized parallel region R_1 that enters another nested serialized parallel region R_2.

– The U version uses a lightweight implementation of serialized parallel regions for speed but lacks special support for introspection consistency. This version is basically upstream LLVM OpenMP[2] adjusted to not provide incorrect information about enclosing serialized parallel regions during creation of an inner region. As a result, it may report that information about R_1 is Unavailable while R_2 is being created.
– The W version, described in the previous section, uses a lightweight implementation of serialized parallel regions and supports introspection consistency by using Wait-free coordination between the LLVM OpenMP runtime and its OMPT introspection routines.
– The F version is modified from the U version to implement serialized parallel regions using Full team descriptors. With our changes to avoid mismatched team and task information for parallel regions described in the Sect. 3, this version supports introspection consistency.
– The B version provides introspection consistency by Blocking all signals during the creation/destruction of a nested serialized region, preventing introspection while state is inconsistent. We modified the U version to block signals using the Linux `sigprocmask` routine while a serialized region is created/destroyed.

We compared the performance of the runtime versions on a system with one Intel Xeon Phi 7250 processor with 68 4-way SMT cores with 115 GB of DRAM running CentOS Linux 7.2.1511. The system was chosen principally because of its availability for isolated experiments. To avoid performance variability caused by different code and data layout in our experiments, we disabled Linux Address Space Layout Randomization. On the system's `x86_64` processors, we implemented the `compare-and-swap` used for wait-free coordination in the W runtime version using the `cmpxchg` instruction. No `lock` prefix is necessary since the `compare-and-swap` coordinates between the OpenMP runtime and a signal handler making introspection calls executed by the same thread.

We conducted two groups of experiments. The first group uses one synthetic microbenchmark, S (Listing 1), designed to measure the worst-case overhead of maintaining nested serialized parallel regions. S contains a serialized parallel region that spawns 16 million trivial nested serialized parallel regions. For the second group of experiments, we used the SPEC OMP 2012 [9] benchmark suite. We compiled all runtime implementations U, W, F, and B with Clang

[2] Forked from the commit with the hash b552adf8b388a4fbdaa6fb46bdedc83fc738fc2b on March 11th 2021.

```
#pragma omp parallel num_threads(1)
  for (int i = 0; i < 16000000; i++)
    #pragma omp parallel num_threads(1)
      volatile int x = 0;
```

Listing 1. Microbenchmark S measures the overhead of creating nested serialized parallel regions.

12.0.0 into four shared libraries in Release mode with -O3 optimization and `OMPT_SUPPORT=on` and used them in both groups of experiments.

5.1 Stress Testing of OpenMP Runtime Variants

The first group of experiments represents a stress testing of the runtime implementations overhead. For this purpose, we compiled S microbenchmark with Clang 12.0.0 using -O3 optimization and -g to provide line maps for tools. We created four executables by dynamically linking S to each of the U, W, F, and B runtime shared libraries. In our experiments, we measured each microbenchmark 30 times, computing its average execution time and standard deviation. For more representative results, the 30 runs of each microbenchmark are performed by three processes, each measuring ten runs of the microbenchmark after a warmup run.

Table 1 presents measurements of the S microbenchmark linked to each of the U, W, F and B runtime versions under three conditions: (a) with no tool present, (b) with a trivial OMPT tool that performs no measurement but causes the runtime maintain OMPT state, and (c) a basic OMPT sampling tool that periodically interrupts the program and invokes the OMPT `ompt_get_task_info` introspection routine to inspect every active parallel region and implicit task.[3] We discuss the performance of our microbenchmark under each of these three conditions to assess the cost of providing introspection consistency. The U variant, which does not support introspection consistency, serves as the baseline for the other three runtime implementations.

No Tool. Running a program without a tool is the common case, so high performance is important. Table 1(a) compares the cost of executing the S microbenchmark for serialized parallel regions using each of the U, W, F, and B runtime variants with no tool present. Overhead for S_W, S_F, and S_B are relative to S_U. The W version, which uses a wait-free protocol to provide introspection consistency was 1.49% faster than the U version which does not support introspection consistency. Our claim here is not that W is inherently faster than U but rather that they have comparable performance with no tool. Although blocking signals does not happen when the tool is not attached, introducing calls to `sigprocmask` changed the code layout resulting in 7% more overhead. One may not observe

[3] HPCToolkit uses `ompt_get_task_info` to assemble user-level calling contexts for all OpenMP work.

Table 1. Performance of benchmark S (Listing 1), which repeatedly executes a trivial nested serialized parallel region using four runtimes: U may report region information Unavailable; W uses a Wait-free strategy to implement introspection consistency; F implements nested parallelism using Full team descriptors; B may Block signals to support introspection consistency.

	(a) No tool		(b) Trivial tool		(c) Sampling Tool	
Code	Time(s)	Ovhd(%)	Time(s)	Ovhd(%)	Time(s)	Ovhd(%)
S_U	8.9846 ± 0.0006	-	10.926 ± 0.004	-	10.959 ± 0.001	0.30
S_W	8.8508 ± 0.0009	-1.49	12.477 ± 0.007	14.20	12.513 ± 0.008	0.29
S_F	16.077 ± 0.007	78.94	16.241 ± 0.012	48.65	16.280 ± 0.002	0.24
S_B	9.656 ± 0.002	7.47	43.965 ± 0.056	302.40	44.062 ± 0.024	0.22

this cost on other machines with more cache per core. In contrast, F, which always allocates and initializes full team descriptors, is almost 80% slower than the others. This shows that the complexity of W's wait-free coordination needed for introspection consistency with optimized serialized parallel regions is a good alternative to F's simpler protocol based on full region descriptors.

Trivial Tool. We developed a simple tool that uses the `ompt_start_tool` function, an initializer, and a finalizer to inform the runtime that a tool is present, but that's all; it doesn't use register for OMPT callbacks or enable sampling. Table 1(b) compares the cost of executing the S microbenchmark for serialized parallel regions using each of the U, W, F, and B runtime variants with a trivial tool, calculating overhead relative to S_U. Compared with the no tool version in column (a), S_U with a tool, which must maintain lwt descriptors, is 21.6% slower. On top of that, the wait-free coordination in the W version adds a 14.2% overhead. This shows that the cost of maintaining the lwt descriptors with a wait-free protocol is about 2/3 more costly than introducing lwt descriptors in the first place. The F version, which does not maintain lwt descriptors, has an overhead of almost 49%, which is more than 3×- higher than the overhead of the W version. Again, W delivers introspection consistency much cheaper than F, which maintains full descriptors for nested serialized regions. Furthermore, blocking signals while profiling short nested parallel regions is extremely costly. Namely, the B version is almost 3.5×- slower than the W version, meaning that providing introspection consistency using wait-free coordination is suitable for short nested parallel regions.

Sampling Tool. To assess the performance of nested serialized parallel regions for the four runtime variants while being observed with a sampling-based OMPT tool, we developed a simple proxy tool for benchmarking. We extended the trivial tool from the previous experiments with a simple signal handler and configured each thread to receive 200 samples per second from a Linux CPUTIME timer. The signal handler calls `ompt_get_task_info` for each available enclosing parallel and task region. Unlike the previous experiments in Table 1(a) and Table 1(b), for the sampling-based measurements in Table 1(c), we calculate the overhead of sampling for each code relative to the trivial tool times in Table 1(b). Each of the runtime versions have similar overhead from sampling, even though for the W version, invoking `ompt_get_task_info` might require additional work to finish assembly of a nested serialized parallel region that was in progress when the runtime was interrupted. Less than 10% of the asynchronous samples were received while the runtime was executing the code that requires wait-free synchronization between the runtime and the introspection routine invoked from a signal handler, which explains why the difference in time between Table 1(b) and Table 1(c) is similar for the U and W versions.

5.2 OpenMP Runtime Performance in Real-World Scenarios

To test the performance of U, W, F, and B runtime implementations in real-world scenarios, we used the SPEC OMP 2012 benchmark suite [9]. We created a configuration file for each runtime shared library to be used by the runspec [9] running tool. All configuration files specify the usage of intel compilers icc/icpc and ifort (version 16.0.3) with -O3 optimization for compiling C/C++ and Fortran benchmarks, respectively. We observed that thread binding sometimes slows a benchmark's execution, so we disabled it by setting `OMP_PROC_BIND` to false. We supply the configuration files to runspec. By default, runspec runs each benchmark three times with the reference workload with no profiling tool attached, meaning OMPT support is compiled but not used. Runspec finds each benchmark's median run time and divides it by the reference system's run time to calculate the normalized ratio. Finally, runspec calculates the geometric mean of all fourteen benchmark normalized ratios.

For brevity, Table 2 provides only the geometric mean of normalized ratios for each runspec invocation supplied with configuration files corresponding to the U, W, F, and B, respectively. The W runtime version employing wait-free coordination shows a negligible drop in performance compared to the others. We observed an overhead of about 3% when running the 376-tree benchmark, which spawns many recursive explicit tasks. Measurements using Linux `perf` showed that W causes more branch and instruction cache misses than U. Although the code we introduced for the wait-free coordination protocol is not executed when creating explicit tasks, it changes the code and data layout, which has a surprising effect on the cache performance on the Xeon Phi.

The W implementation outperforms the F and B versions while providing introspection consistency for short nested serialized parallel regions. However, the results presented in Table 2 show that nested serialized parallel regions are not widespread in real-world applications.

6 Related Work

The OMPT interface has been widely adopted by open-source performance tools including Caliper [4], HPCToolkit [12], Tau [11], and Score-P [8] as well as data race detection tools such as ARCHER [2], ROMP [6], and SWORD [3].

Table 2. The final measurements reports of SPEC OMP 2012 benchmark suite run four times with each of the U, W, F, and B runtime implementation. One run of suite assumes running all 14 benchmarks link to the same runtime version, determining the median run time, dividing it by the reference time to calculate normalized ratio, and calculating the geometric mean of all 14 ratios.

Runtime	Geomean
$SPEC_U$	4.23
$SPEC_W$	4.22
$SPEC_F$	4.23
$SPEC_B$	4.23

With the exception of Tau and HPCToolkit, which support asynchronous sampling, the remainder of these tools use OMPT callbacks and synchronous calls to introspection routines. Only tools that monitor OpenMP programs with asynchronous sampling are affected when OpenMP implementations lack support for introspection consistency.

7 Conclusions

An OpenMP implementation that supports introspection consistency for parallel and task regions is necessary for sampling-based performance tools to provide accurate information about nested regions. We have described strategies for supporting introspection consistency, including how to coordinate entry to regular parallel regions and a wait-free coordination protocol for nested serialized parallel regions, which efficiently handles a corner case that was an impediment to introspection consistency.

Our experiments with the microbenchmark that stresses the runtime implementation to its limits have shown that the cost of providing introspection consistency is negligible without a tool. When sampling is enabled, our wait-free implementation of optimized serialized parallel regions delivers introspection consistency at a significantly lower cost than allocating and initializing full region team descriptors or blocking signals to provide introspection consistency. We found that the runtime overhead for providing introspection consistency in a representative set of HPC benchmarks is negligible. The drawback of our approach is the complexity introduced to handle a corner case not commonly encountered in OpenMP applications. This might encourage runtime developers to prefer an alternative strategy such as blocking signals while entering or leaving a nested serialized parallel region.

In our view, the benefit of introspection consistency for tools greatly outweighs its cost. As a result, we believe that the next OpenMP standard should specify that OpenMP implementations and their OMPT introspection routines must support introspection consistency to be standard-conforming. It is worth noting that the OpenMP Debugging API [10] also needs introspection consistency. Since one can interrupt a program execution at any time in a debugger, the OMPD interface would benefit from being able to determine the nesting of parallel and task regions at arbitrary points in time using mechanisms described in this paper.

Although we developed a wait-free coordination protocol to solve a problem specific to the LLVM OpenMP runtime implementation, our approach is more broadly applicable. Namely, whenever a program manipulates data that a signal handler can inspect and change at any time, our wait-free coordination approach handles the data race between the runtime and a signal handler.

References

1. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010)
2. Atzeni, S., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62 (2016)
3. Atzeni, S., et al.: SWORD: a bounded memory-overhead detector of OpenMP data races in production runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 845–854 (2018)
4. Boehme, D., et al.: Caliper: performance introspection for HPC software stacks. In: SC 2016: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 550–560 (2016)
5. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
6. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: SC 2018: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 767–778 (2018)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Programm. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
8. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M., Nagel, W., Resch, M. (eds.) Tools for High Performance Computing, pp. 79–91. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-31476-6_7
9. Müller, M.S., et al.: SPEC OMP2012—an application benchmark suite for parallel systems using OpenMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 223–236. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_17
10. OpenMP Architecture Review Board: OpenMP application programming interface, version 5.0 (2018)

11. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Intl. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
12. Tallent, N.R., et al.: Scalable fine-grained call path tracing. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 63–74. Association for Computing Machinery, New York (2011)
13. Zhou, K., et al.: Measurement and analysis of GPU-accelerated applications with HPCToolkit. Parallel Comput. **108**, 102837 (2021)