OPENMP ARB

OpenMP Technical Report 2 on the OMPT Interface

This Technical Report specifies OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis

Alexandre Eichenberger (co-editor, alexe@us.ibm.com)
John Mellor-Crummey (co-editor, johnmc@rice.edu)
Martin Schulz (co-editor, schulzm@lnl.gov)

Nawal Copty
Jim Cownie
Robert Dietrich
Xu Liu
Eugene Loh
Daniel Lorenz
and other members of the OpenMP Tools Working Group

4/3/2014

Expires: 4/3/2017

We actively solicit comments. Please provide feedback on this document either to the Editor directly or in the OpenMP Forum at openmp.org

End of Public Comment Period: June 2, 2014

OpenMP Architecture Review Board www.openmp.org info@openmp.org
c/o David K. Poulsen, OpenMP, 1906 Fox Drive, Champaign, Illinois 61820 USA

This technical report describes possible future directions or extensions to the OpenMP Specification.

The goal of this technical report is to build more widespread existing practice for an expanded OpenMP. It gives advice on extensions or future directions to those vendors who wish to provide them possibly for trial implementation, allows OpenMP to gather early feedback, support timing and scheduling differences between official OpenMP releases, and offers a preview to users of the future directions of OpenMP with the provision stated in the next paragraph.

This technical report is non-normative. Some of the components in this technical report may be considered for standardization in a future version of OpenMP, but they are not currently part of any OpenMP Specification. Some of the components in this technical report may never be standardized, others may be standardized in a substantially changed form, or it may be standardized as is in its entirety.

OMPT: An OpenMP® Tools Application Programming Interface for Performance Analysis

Alexandre Eichenberger[*], John Mellor-Crummey[†], Martin Schulz[‡],

Nawal Copty[§], Jim Cownie[¶], Robert Dietrich[‖], Xu Liu[†], Eugene Loh[§], Daniel Lorenz[**]
and other members of the OpenMP Tools Working Group

March 21, 2014

1 Introduction

Today, it is difficult to produce high quality tools that support performance analysis of OpenMP programs without tightly integrating them with a specific OpenMP runtime implementation. To address this problem, this document defines OMPT—an application programming interface (API) for first-party performance tools.[1] Extending the OpenMP standard with this API will make it possible to construct powerful tools that will support any standard-compliant OpenMP implementation.

1.1 OMPT

The design of OMPT is based on experience with two prior efforts to define a standard OpenMP tools API: the POMP API [**?**] and the Sun/Oracle Collector API [**?**, **?**]. The POMP API provides support for instrumentation-based measurement. A drawback of this approach is that its overhead can be significant because an operation, e.g., an iteration of an OpenMP worksharing loop, may take less time than tool callbacks monitoring its execution. In contrast, the Sun/Oracle Collector API was designed primarily to support performance measurement using asynchronous sampling. This design enables the construction of tools that attribute costs without the overhead and intrusion of pervasive instrumentation. With the Collector API, tools can use low-overhead asynchronous sampling of application call stacks to record compact call path profiles. However, the Collector API doesn't provide enough instrumentation hooks to provide full tool support for statically-linked executables. OMPT builds upon ideas from both the POMP and Collector APIs. The core of OMPT is a minimal set of features to support tools that employ asynchronous sampling to measure application performance. In addition, OMPT defines interfaces to support *blame shifting* [**?**, **?**]—a technique that shifts attribution of costs from symptoms to causes. Finally, OMPT defines callbacks suitable for instrumentation-based monitoring of runtime events. OMPT can be implemented entirely by a compiler, entirely by an OpenMP runtime system, or with a hybrid strategy that employs a mixture of compiler and runtime support.

With the exception of one routine for tool control, all functions in the OMPT API are intended for use only by tools rather than by applications. All OMPT API functions have a C binding. A Fortran binding is provided only for the single application-facing tool control function described in Section **??**.

[*]IBM T.J. Watson Research Center
[†]Rice University
[‡]Lawrence Livermore National Laboratory
[§]Oracle
[¶]Intel
[‖]TU Dresden, ZIH
[**]Jülich Supercomputer Center

[1]A *first-party* tool runs within the address space of an application process. This differs from a *third-party* tool, e.g., a debugger, which runs as a separate process.

1.1.1 Design Objectives

OMPT tries to satisfy several design objectives for a performance tool interface for OpenMP. These objectives are listed in decreasing order of importance.

- The API should enable tools to gather sufficient information about an OpenMP program execution to associate costs with both the program and the OpenMP runtime system.
  - The API should provide an interface sufficient to construct low-overhead performance tools based on asynchronous sampling.
  - The API should enable a profiler that uses call stack unwinding to identify which frames in its call stack are present on behalf of the OpenMP runtime.
  - An OpenMP runtime system should associate the activity of a thread at any point in time with a *state*, e.g., idle, which will enable a performance tool to interpret program behavior.
  - Certain API routines must be defined as *async signal safe* so that they can be invoked in a profiler's signal handler as it processes interrupts generated by asynchronous sampling.
- Incorporating support for the API in an OpenMP runtime system should add negligible overhead to the runtime system if the interface is not in use by a tool.
- The API should define interfaces suitable for constructing instrumentation-based performance tools.
- Adding the API to an OpenMP runtime should not impose an unreasonable development burden on the runtime developer.
- The API should not impose an unreasonable development burden on tool implementers.

To support the OMPT interface for tools, an OpenMP runtime system must maintain information about the state of each OpenMP thread and provide a set of API calls that tools can use to interrogate the OpenMP runtime. Maintaining information about the state of each thread in the runtime system is not free and thus an OpenMP runtime system need not maintain state information unless a tool has registered its interest in this information. Without any explicit request to enable tool support, an OpenMP runtime need not maintain any state for the benefit of tools.

1.1.2 Minimally Compliant Implementation

OMPT has a small set of mandatory features that provide a common foundation for all performance tools. A runtime may also implement additional, optional, OMPT features used by some tools to gather extra information about a program execution. The features required by a minimally compliant implementation are summarized below.

- Maintain a unique numerical ID per OpenMP thread, parallel region, and task region. A minimal implementation may reuse the task ID required by OpenMP for nested locks.
- Maintain pointers into the stack for each OpenMP thread to distinguish frames for user procedures from frames for OpenMP runtime routines. Each OpenMP worker must maintain a pointer to the stack frame of the runtime routine containing its idle loop, if one exists on the stack.
- Maintain a state and a wait condition for each OpenMP thread. Mandatory states are idle, work serial, work parallel, and undefined.
- Provide callbacks to tools when encountering the following events: thread begin/end, parallel region begin/end, task region begin/end, a user-level tool control call, and runtime shutdown.
- Implement several async signal safe inquiry functions to retrieve information from the OpenMP runtime.
- Have the OpenMP runtime initiate a callback to a tool initialization routine as directed by the value of a new OpenMP environment variable (`OMP_TOOL`) and provide a function to register tool callbacks with the runtime.

1.2 Document Roadmap

This document first outlines various aspects of the OMPT tools API. Section **??** describes the state information maintained by the OpenMP runtime system on behalf of OMPT for use by tools. Section **??** describes the OMPT callbacks to notify a tool of various OpenMP runtime events during an execution. Section **??** describes the data structures used by the OMPT interface. Section **??** describes the runtime system inquiry operations supported by OMPT for the benefit of tools. Section **??** describes the OMPT API operations for tool initialization. Section **??** describes the tool control interface available to applications. Section **??** concludes with a few notes about potential future enhancements. Appendix **??** provides a definition of the complete OMPT interface in C. Appendix **??** illustrates the information that OMPT maintains about call stacks and the use of OMPT API routines to inspect it; this support enables tools to associate code executed in OpenMP parallel regions with application-level calling contexts.

2 Runtime States

To enable a tool to understand what an OpenMP thread is doing, when a tool registers itself with an OpenMP runtime system, the runtime will maintain state information for each OpenMP thread that can be queried by the tool. The state maintained for each thread by the OpenMP runtime is an approximation of the thread's instantaneous state. OMPT uses the enumeration type `ompt_state_t` for states; Appendix **??** defines this type. When the state of a thread not associated with the OpenMP runtime is queried, the runtime returns `ompt_state_undefined`.

For each OpenMP thread the runtime maintains not only a state but also an `ompt_wait_id_t` identifier. When a thread is waiting for a lock, critical region, ordered, or atomic, and the thread is in a wait state, then the thread's `wait_id` field identifies the lock, critical construct, ordered construct, atomic construct, or internal variable upon which the thread is waiting. The semantics of the values used for a `wait_id` are implementation defined. A thread's `wait_id` is undefined if the thread is not in a wait state.

States are classified as *mandatory*, *optional*, or *flexible*. Flexible states provide an OpenMP runtime with leeway to determine if and when to report transitions to a flexible state. For example, consider when a thread acquires a lock. One compliant runtime may transition a thread's state to `ompt_state_wait_lock` (flexible state) early before the thread attempts to acquire a lock. Another compliant runtime may transition a thread's state to `ompt_state_wait_lock` late, only if the thread begins to spin or block to wait for an unavailable lock. A third compliant runtime may transition a thread's state to `ompt_state_wait_lock` even later, e.g., only after the thread waits for a significant amount of time.

State values 0 to 127 are reserved for current OMPT states and future extensions.

Idle State

`ompt_state_idle` (mandatory)

The thread is idle while waiting to work on an OpenMP parallel region.

Work States

`ompt_state_work_serial` (mandatory)

The thread is executing code outside all parallel regions.

`ompt_state_work_parallel` (mandatory)

The thread is executing code within the scope of a parallel region construct.

`ompt_state_work_reduction` (optional)

The thread is combining partial reduction results from threads in its team. A compliant runtime might never report a thread in this state; a thread combining partial reduction results may report its state as `ompt_state_work_parallel` or `ompt_state_overhead`.

Barrier Wait States

`ompt_state_wait_barrier` (flexible)

The thread is waiting at either an implicit or explicit barrier. A compliant implementation may have a thread enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. A compliant implementation may never report a thread in this state; instead a thread might report its state as `ompt_state_wait_barrier_implicit` or `ompt_state_wait_barrier_explicit`, as appropriate.

`ompt_state_wait_barrier_implicit` (flexible)

The thread is waiting at an implicit barrier in a parallel region. A compliant implementation may have a thread enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. A compliant runtime implementation may report `ompt_state_wait_barrier` for implicit barriers.

`ompt_state_wait_barrier_explicit` (flexible)

The thread is waiting at an explicit barrier in a parallel region. A compliant implementation may have a thread enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. A compliant runtime implementation may report `ompt_state_wait_barrier` for explicit barriers.

Task Wait States

`ompt_state_wait_taskwait` (flexible)

The thread is waiting at a taskwait construct. A compliant implementation may have a thread enter this state early, when the thread encounters a taskwait construct, or late, when the thread begins to wait for an uncompleted task.

`ompt_state_wait_taskgroup` (flexible)

The thread is waiting at the end of a taskgroup construct. A compliant implementation may have a thread enter this state early, when the thread encounters the end of a taskgroup construct, or late, when the thread begins to wait for an uncompleted task.

Mutex Wait States

`ompt_state_wait_lock` (`ompt_state_wait_nest_lock`) (flexible)

The thread is waiting for a lock (nest lock). A compliant implementation may have a thread enter this state early, when a thread encounters a lock (nest lock) `set` routine, or late, when the thread begins to wait for a lock (nest lock).

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to identify the lock (nest lock) being acquired.

`ompt_state_wait_critical` (flexible)

The thread is waiting to enter a critical region. A compliant implementation may have a thread enter this state early, when the thread encounters a critical construct, or late, when the thread begins to wait to enter the critical region. A compliant implementation may report a thread waiting to enter a critical region in `ompt_state_wait_lock` if waiting for a lock associated with the construct.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to identify the critical construct or an internal runtime variable (e.g., a lock) associated with the critical construct.

`ompt_state_wait_atomic` (flexible)

The thread is waiting to enter an atomic region. A compliant implementation may have a thread enter this state early, when the thread encounters an atomic construct, or late, when the thread begins to wait to enter the atomic region. A compliant implementation may report a thread waiting to enter an atomic region in `ompt_state_wait_lock` if waiting for a lock associated with the atomic construct. A compliant implementation may opt to not report this state, for example, when using atomic hardware instructions, which allow non-blocking atomic implementations.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to identify the atomic construct, a program variable, or an internal runtime variable (e.g., a lock) associated with the atomic construct.

`ompt_state_wait_ordered` (flexible)

The thread is waiting to enter an ordered region. A compliant implementation may have a thread enter this state early, when the thread encounters an ordered construct, or late, when the thread begins to wait to enter the ordered region. A compliant implementation may report a thread waiting to enter a ordered region in `ompt_state_wait_lock` if waiting for a lock associated with the ordered construct.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to identify the ordered construct or an internal runtime variable (e.g., a lock) associated with the ordered construct.

Overhead State

`ompt_state_overhead` (optional)

A thread may be reported as being in the overhead state at any point while executing within an OpenMP runtime system, e.g., while preparing a parallel region, preparing a new explicit task, preparing a worksharing region, or preparing to execute iterations of a parallel loop. It is compliant to report some or all OpenMP runtime overhead as work.

Miscellaneous States

`ompt_state_undefined` (mandatory)

This state is reserved for threads that are not user threads, initial threads, threads currently in an OpenMP team, or threads waiting to become part of an OpenMP team.

`ompt_state_first` (mandatory)

This state is a placeholder exclusively reserved for use by the OMPT runtime call `ompt_enumerate_state` (see Section **??**), which is used to enumerate all available runtime states. A thread will never be reported in this state.

3 Events

This section describes callback events that an OpenMP runtime may provide for use by a tool. OMPT uses the enumeration type `ompt_event_t` for events; Appendix **??** defines this type. A tool need not register a callback for any particular event. In most cases, an OpenMP runtime system will not make any callback unless a tool has registered to receive it. The exception to this rule is begin/end event pairs. To implement event notifications efficiently, a runtime may assume that for certain begin/end event pairs if one event of the pair has a callback registered, the other will have a callback defined as well. When this exception applies, it will be noted for affected events.

Callbacks for different events may have different type signatures. The type signature for an event's callback is noted with the event definition. Appendix **??** defines type signatures for callback events.

There are two classes of events: mandatory events and optional events. Mandatory events must be implemented in any compliant OpenMP runtime implementation. Optional events are grouped in sets of related events. Except for begin/end pairs as noted, support for any particular optional event can be included or omitted at the discretion of a runtime system implementer.

3.1 Mandatory Events

The following callback events are mandatory and must be supported by a compliant OpenMP runtime system.

Threads

`ompt_event_thread_begin`

The OpenMP runtime invokes this callback in the context of an initial thread just after it initializes the OpenMP runtime for itself, or in the context of a new thread created by the OpenMP runtime system just after the thread initializes itself. In either case, this callback must be the first callback for a thread and must occur before the thread executes any OpenMP tasks. The type of the thread (`ompt_thread_initial`, `ompt_thread_worker`, or `ompt_thread_other`) is passed as an argument to the callback. This callback has type signature `ompt_thread_type_callback_t`.

`ompt_event_thread_end`

The OpenMP runtime invokes this callback after an OpenMP thread completes all of its tasks but before the thread is destroyed. The callback executes in the context of the OpenMP thread. This callback must be the last callback event for any thread of type `ompt_thread_worker`; it is optional for other types of threads. This callback has type signature `ompt_thread_type_callback_t`.

Parallel Regions

`ompt_event_parallel_begin`

The OpenMP runtime invokes this callback after a task encounters a parallel construct but before any implicit task starts to execute the parallel region's work. The callback executes in the context of the task that encountered the parallel construct. This callback has type signature `ompt_new_parallel_callback_t`, and includes a parameter that indicates the number of threads requested by the user. A tool may use this value as an upper bound on the number of threads that will participate in the team.

`ompt_event_parallel_end`

The OpenMP runtime invokes this callback after a parallel region executes its closing synchronization barrier but before resuming execution of the parent task. The callback executes in the context of the task that encountered the parallel construct. This callback has type signature `ompt_parallel_callback_t`.

Tasks

`ompt_event_task_begin`

The OpenMP runtime invokes this callback after a task encounters a task construct but before the new explicit task executes. The callback executes in the context of the task that encountered the task construct. This callback has type signature `ompt_new_task_callback_t`.

`ompt_event_task_end`

The OpenMP runtime invokes this callback after an explicit task completes but before the thread resumes execution of another task. The callback executes in the context of an arbitrary task on the thread that completed the explicit task. This callback has type signature `ompt_task_callback_t`.

Application Tool Control

`ompt_event_control`

If the user program calls `ompt_control`, the OpenMP runtime invokes this callback. The callback executes in the environment of the user control call; the arguments passed to the callback are the values passed by the user to `ompt_control`. This callback has type signature `ompt_control_callback_t`.

Termination

`ompt_event_runtime_shutdown`

The OpenMP runtime system invokes this callback before it shuts down the runtime system. This callback enables a tool to clean up its state and record or report its measurement data, as appropriate. A runtime may later restart and reinitialize the tool by calling the tool initializer function (`ompt_initialize`, described in Section **??**) again. This callback has type signature `ompt_callback_t`.

3.2 Optional Events

This section describes two sets of events. The first set of events is intended primarily for use by sampling-based performance tools that employ a strategy known as *blame shifting* to attribute waiting to activity in contexts that cause other threads to wait rather than contexts in which waiting is observed. The second set of events, in combination with other mandatory and optional events, enables instrumentation-based tools to receive notification for any or all OpenMP runtime events as they occur.

Support for these events is optional. An OpenMP runtime system remains compliant even if it supports none of the events in this section.

3.2.1 Events for Blame Shifting (Optional)

This section describes callback events used by sampling-based performance tools that employ *blame shifting* to transfer blame for waiting from contexts where waiting is observed to contexts responsible for the waiting.[2] Using these callbacks, a tool employing blame shifting can attribute time that a thread spends waiting for a lock to the context of the lock holder. Similarly, time that threads spend waiting at a barrier can be attributed back to code being executed by working threads while other threads wait.

The events listed immediately below are used by an OpenMP runtime to notify a tool when various kinds of idling begin and end. Since idling indicates the absence of any activity, a thread will not receive any event notification between begin and end notifications for idling.

Idle State Entry/Exit

`ompt_event_idle_begin`

The OpenMP runtime invokes this callback when a thread starts to idle outside a parallel region. The callback executes in the environment of the idling thread. This callback has type signature `ompt_thread_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_idle_end` *must also be registered.*

`ompt_event_idle_end`

The OpenMP runtime invokes this callback when a thread finishes idling outside a parallel region. The callback executes in the environment of the thread that is about to resume useful work. This callback has type signature `ompt_thread_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_idle_begin` *must also be registered.*

[2]The utility of blame shifting has previously been demonstrated for attributing idling while waiting to steal work in a work-stealing runtime [**?**], and spin waiting to acquire a lock [**?**].

Barrier Idling

`ompt_event_wait_barrier_begin`

The OpenMP runtime invokes this callback when an implicit task starts to wait in a barrier region. One barrier region may generate multiple pairs of barrier begin and end callbacks in a task, e.g., if waiting at the barrier occurs in multiple stages or if another task is scheduled on this thread while it waits at the barrier. The callback executes in the context of an implicit task waiting for a barrier region to complete. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_barrier_end` *must also be registered.*

`ompt_event_wait_barrier_end`

The OpenMP runtime invokes this callback when an implicit task finishes waiting in a barrier region. One barrier region may generate multiple pairs of barrier begin and end callbacks in a task, e.g., if waiting at the barrier occurs in multiple stages or if another task is scheduled on this thread while it waits at the barrier. The callback executes in the context of an implicit task waiting for a barrier region to complete. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_barrier_begin` *must also be registered.*

Taskwait Idling

`ompt_event_wait_taskwait_begin`

The OpenMP runtime invokes this callback when a thread starts to wait in a taskwait region. One taskwait region may generate multiple pairs of taskwait begin and end callbacks if another task is scheduled on this thread while it waits at the taskwait. This callback executes in the context of the task that encountered the taskwait construct. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_taskwait_end` *must also be registered.*

`ompt_event_wait_taskwait_end`

The OpenMP runtime invokes this callback when a task finishes waiting in a taskwait region. One taskwait region may generate multiple pairs of taskwait begin and end callbacks if another task is scheduled on this thread while it waits in the taskwait region. This callback executes in the context of the task that encountered the taskwait construct. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_taskwait_begin` *must also be registered.*

Taskgroup Idling

`ompt_event_wait_taskgroup_begin`

The OpenMP runtime invokes this callback when a task starts to wait for a taskgroup region to complete. One taskgroup region may generate multiple pairs of taskgroup begin and end callbacks if another task is scheduled on this thread while it waits in the taskgroup region. This callback executes in the context of the task that encountered the taskgroup construct. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_taskgroup_end` *must also be registered.*

`ompt_event_wait_taskgroup_end`

The OpenMP runtime invokes this callback when a task finishes waiting for a taskgroup region to complete. One taskgroup region may generate multiple pairs of taskgroup begin and end callbacks if another task is scheduled on this thread while it waits in the taskgroup region. This callback executes in the context of the task that encountered the taskgroup construct. This callback has type signature `ompt_parallel_callback_t`.

Note: If this callback is registered, the callback for `ompt_event_wait_taskgroup_begin` *must also be registered.*

Lock Release

`ompt_event_release_lock`

The OpenMP runtime system invokes this callback after a task releases a lock. This callback executes in the context of the task that called `omp_unset_lock`; its `wait_id` parameter identifies the released lock. This callback has type signature `ompt_wait_callback_t`.

Note: This callback may be useful to an instrumentation-based tool to terminate an interval beginning with `ompt_event_acquired_lock`.

`ompt_event_release_nest_lock_last`

The OpenMP runtime invokes this callback for certain releases of a nest lock. If a task acquires a nest lock n times, this callback occurs only after the n^{th} release. The inner $n-1$ releases are reported as `ompt_event_release_nest_lock_prev` events. This callback executes in the context of the task that called `omp_unset_nest_lock`; its `wait_id` parameter identifies the nest lock released. This callback has type signature `ompt_wait_callback_t`.

Note: This callback may be useful to an instrumentation-based tool to terminate an interval beginning with `ompt_event_acquired_nest_lock_first`.

Critical Release

`ompt_event_release_critical`

The OpenMP runtime system invokes this callback after a task exits a critical region. This callback executes in the context of the task that encountered the critical construct; its `wait_id` parameter identifies the critical construct or an internal runtime variable (e.g., a lock) associated with the critical construct that was exited. This callback has type signature `ompt_wait_callback_t`.

Note: This callback may be useful to an instrumentation-based tool to terminate an interval beginning with `ompt_event_acquired_critical`.

Ordered Release

`ompt_event_release_ordered`

The OpenMP runtime system invokes this callback after a task exits an ordered region. This callback executes in the context of the task that encountered the ordered construct; its `wait_id` parameter identifies the ordered construct or an internal runtime variable (e.g., a lock) associated with the ordered construct that was exited. This callback has type signature `ompt_wait_callback_t`.

Note: This callback may be useful to an instrumentation-based tool to terminate an interval beginning with `ompt_event_acquired_ordered`.

Atomic Release

`ompt_event_release_atomic`

The OpenMP runtime system invokes this callback after a task completes an atomic region. This callback executes in the context of the task that encountered the atomic construct; its `wait_id` parameter identifies the atomic construct, a program variable, or an internal runtime variable (e.g., a lock) associated with the atomic construct being exited.

If an atomic block is implemented using a hardware instruction, then an OpenMP runtime may choose never to report this event. However, if an atomic region is implemented using any mechanism that involves a software protocol that spin waits or retries, then an OpenMP runtime developer should consider reporting this event to accept blame for any spin waiting or retries that the atomic region causes. Examples of spinning in software include spin waiting for a critical region used to implement atomics, or retrying atomic operations implemented using hardware primitives that may fail. Examples of hardware primitives that could fail with explicit retries in software include transactional instructions, load-linked/store-conditional, or compare-and-swap.

This callback has type signature `ompt_wait_callback_t`.

Note: This callback may be useful to an instrumentation-based tool to terminate an interval beginning with `ompt_event_acquired_atomic`*.*

3.2.2 Events for Instrumentation-based Measurement Tools (Optional)

The following set of events, in combination with other mandatory and optional events, enables instrumentation-based tools to receive notification for any or all OpenMP runtime events as they occur.

Task Creation and Destruction

`ompt_event_implicit_task_begin`

The OpenMP runtime system invokes this callback, after an implicit task is fully initialized but before the task executes its work. This callback executes in the context of the new implicit task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_implicit_task_end`

The OpenMP runtime system invokes this callback after an implicit task executes its closing synchronization barrier but before returning to idle or the task is destroyed. The callback executes in the context of the implicit task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_initial_task_begin`

The OpenMP runtime system invokes this callback, just after an initial implicit task is fully initialized but before it starts to execute. This callback executes in the context of an initial task. This callback has type signature `ompt_task_callback_t`.

`ompt_event_initial_task_end`

The OpenMP runtime system invokes this callback when an initial implicit task ends but before the task is destroyed. The callback executes in the context of the initial implicit task. This callback has type signature `ompt_task_callback_t`,

`ompt_event_task_switch`

The OpenMP runtime system invokes this callback after it suspends one task and before it resumes another task. This callback executes in the context of the resumed task. If the suspended task actually completed and its data structure was deallocated, the value of the `suspended_task_id` parameter is 0. This callback has type signature `ompt_task_switch_callback_t`.

Lock Creation and Destruction

`ompt_event_init_lock` (`ompt_event_init_nest_lock`)

The OpenMP runtime system invokes this callback just after a task initializes a lock (nest lock). This callback executes in the context of the task that called `omp_init_lock` (`omp_init_nest_lock`); its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_destroy_lock` (`ompt_event_destroy_nest_lock`)

The OpenMP runtime system invokes this callback just before a task destroys a lock (nest lock). This callback executes in the context of the task that called `omp_destroy_lock` (`omp_destroy_nest_lock`); its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

Loops

`ompt_event_loop_begin`

The OpenMP runtime system invokes this callback after a task encounters a loop construct but before the task executes its first iteration of the loop. This callback executes in the context of the task that encountered the loop. This callback has type signature `ompt_new_workshare_callback_t`.

`ompt_event_loop_end`

The OpenMP runtime system invokes this callback after a task executes its last iteration of a loop region but before the task executes the loop barrier (wait) or the statement following the loop (nowait). This callback executes in the context of the task that encountered the loop. This callback has type signature `ompt_parallel_callback_t`.

Sections

`ompt_event_sections_begin`

The OpenMP runtime system invokes this callback before a task executes its first section in a sections region. This callback executes in the context of the task that encountered the sections construct. This callback has type signature `ompt_new_workshare_callback_t`.

`ompt_event_sections_end`

The OpenMP runtime system invokes this callback after a task executes its last section in a sections region and before the task executes the section barrier (wait) or the statement following the section construct (nowait). This callback executes in the context of the task that encountered the sections construct. This callback has type signature `ompt_parallel_callback_t`.

Single Blocks

`ompt_event_single_in_block_begin`

The OpenMP runtime system invokes this callback after a task encounters a single construct but before it executes the code block of the single construct. This callback executes in the context of the task that will execute the single code block. This callback has type signature `ompt_new_workshare_callback_t`.

`ompt_event_single_in_block_end`

The OpenMP runtime system invokes this callback after a task executes the code block of a single construct but before the task executes the single barrier (wait) or the statement following the single construct (nowait). This callback executes in the context of the task that executed the single code block. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_single_others_begin`

The OpenMP runtime system invokes this callback when a task that encounters a single construct is not chosen to execute the single code block. This callback executes in the context of the task that encountered the single construct. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_single_others_end`

The OpenMP runtime system invokes this callback in a task after that task reports event `ompt_event_single_others_begin` but before the task executes the single barrier (wait) or the statement following the single construct (nowait). This callback executes in the context of the task that encountered the single construct. This callback has type signature `ompt_parallel_callback_t`.

Workshares

`ompt_event_workshare_begin`

The OpenMP runtime system invokes this callback after a task encounters a workshare construct but before the task executes its first unit of work for the workshare. This callback executes in the context of the task that encountered the workshare construct. This callback has type signature `ompt_new_workshare_callback_t`.

`ompt_event_workshare_end`

The OpenMP runtime system invokes this callback after a task executes its last unit of work for a workshare and before the task executes the workshare barrier (wait) or the statement following the workshare construct (nowait). This callback executes in the context of the task that encountered the workshare construct. This callback has type signature `ompt_parallel_callback_t`.

Master Blocks

`ompt_event_master_begin`

The OpenMP runtime system invokes this callback after the implicit task of a master thread encounters a master construct but before the task executes the master region. This callback executes in the context of the master task of a team. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_master_end`

The OpenMP runtime system invokes this callback after the implicit task of a master thread executed the master region but before the task executes the statement following the master construct. This callback executes in the context of the master task of a team. This callback has type signature `ompt_parallel_callback_t`.

Barriers

`ompt_event_barrier_begin`

The OpenMP runtime system invokes this callback before an implicit task begins execution of a barrier region. This callback executes in the context of the implicit task that encountered the barrier construct. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_barrier_end`

The OpenMP runtime system invokes this callback after an implicit task exits a barrier region. This callback executes in the context of the implicit task that encountered the barrier construct. This callback has type signature `ompt_parallel_callback_t`.

Taskwait

`ompt_event_taskwait_begin`

The OpenMP runtime system invokes this callback after a task encounters a taskwait construct but before the task begins execution of the taskwait region. This callback executes in the context of the task that encountered the taskwait construct. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_taskwait_end`

The OpenMP runtime system invokes this callback after a task exits a taskwait region. This callback executes in the context of the task that encountered the taskwait construct. This callback has type signature `ompt_parallel_callback_t`.

Taskgroup

`ompt_event_taskgroup_begin`

The OpenMP runtime system invokes this callback before a task begins execution of a taskgroup region. This callback executes in the context of the task that encountered the taskgroup construct. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_taskgroup_end`

The OpenMP runtime system invokes this callback after a task exits a taskgroup region. This callback executes in the context of the task that encountered the taskgroup construct. This callback has type signature `ompt_parallel_callback_t`.

Locks

`ompt_event_wait_lock`

The OpenMP runtime system invokes this callback if a task enters the `ompt_state_wait_lock` state. This callback executes in the context of the task that called `omp_set_lock`; its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_lock`

The OpenMP runtime system invokes this callback just after a task acquires a lock. This callback executes in the context of the task that called `omp_set_lock`; its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

Nest Locks

`ompt_event_wait_nest_lock`

The OpenMP runtime system invokes this callback when a task enters the `ompt_state_wait_nest_lock` state. This callback executes in the context of the task that called `omp_set_nest_lock`; its `wait_id` parameter identifies the nest lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_nest_lock_first`

The OpenMP runtime system invokes this callback just after a task acquires a nest lock for the first time. This callback executes in the context of the task that called `omp_set_nest_lock`; its `wait_id` parameter identifies the nest lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_release_nest_lock_prev`

The OpenMP runtime system invokes this callback after a task releases a nest lock that is still owned by this task after the release. If a nest lock was acquired n times by the same task, this callback occurs for the inner $n-1$ releases. The n^{th} release is handled by the `ompt_event_release_nest_lock_last` event. This callback executes in the context of the task that called `omp_unset_nest_lock`; its `wait_id` parameter identifies the nest lock unset. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_nest_lock_next`

The OpenMP runtime system invokes this callback just after this task acquires a nest lock that was already owned by this task. This callback executes in the context of the task that called `omp_unset_nest_lock`; its `wait_id` parameter identifies the nest lock set. This callback has type signature `ompt_wait_callback_t`.

Critical Sections

`ompt_event_wait_critical`

The OpenMP runtime system invokes this callback when this task enters the `ompt_state_wait_critical` state. This callback executes in the context of the task that encountered the critical construct; its `wait_id` parameter identifies the critical construct. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_critical`

The OpenMP runtime system invokes this callback just after this task enters a critical region. This callback executes in the context of the task that encountered the critical construct; its `wait_id` parameter identifies the critical region being entered. This callback has type signature `ompt_wait_callback_t`.

Ordered Sections

`ompt_event_wait_ordered`

The OpenMP runtime system invokes this callback when this task enters the `ompt_state_wait_ordered` state. This callback executes in the context of the task that encountered the ordered construct; its `wait_id` parameter identifies the ordered construct. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_ordered`

The OpenMP runtime system invokes this callback just after this task enters an ordered region. This callback executes in the context of the task that encountered the ordered construct; its `wait_id` parameter identifies a variable associated with the ordered construct. This callback has type signature `ompt_wait_callback_t`.

Atomic Blocks

`ompt_event_wait_atomic`

The OpenMP runtime system invokes this callback when this task enters the `ompt_state_wait_atomic` state. This callback executes in the context of the task that encountered the atomic construct; its `wait_id` parameter identifies the atomic construct, a program variable, or an internal runtime variable (e.g., a lock) associated with the atomic construct being awaited. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_atomic`

The OpenMP runtime system invokes this callback just after this task enters an atomic region. This callback executes in the context of the task that encountered the atomic construct; its `wait_id` parameter identifies the atomic construct, a program variable, or an internal runtime variable (e.g., a lock) associated with the atomic construct being awaited. This callback has type signature `ompt_wait_callback_t`.

Miscellaneous

`ompt_event_flush`

The OpenMP runtime system invokes this callback just after performing a flush operation. This callback executes in the context of the task that encountered the flush construct. This callback has type signature `ompt_thread_callback_t`.

4 Tool Data Structures

4.1 Thread Identifier (Mandatory)

Each OpenMP thread has an associated `ompt_thread_id_t` that uniquely identifies the thread.

```
typedef uint64_t ompt_thread_id_t;
```

The `ompt_thread_id_t` is unique across all thread instances. A OpenMP thread is assigned an ID when the thread begins. A thread's ID is passed to callbacks associated with the begin/end of the thread. A thread ID can be retrieved on demand by invoking the `ompt_get_thread_id` function (described in Section **??**). Tools should not assume that `ompt_thread_id_t` values are consecutive or small. The value 0 is reserved to indicate an invalid thread id.

4.2 Parallel Region Identifier (Mandatory)

Each OpenMP parallel region has an associated `ompt_parallel_id_t` that uniquely identifies the region.

```
typedef uint64_t ompt_parallel_id_t;
```

The `ompt_parallel_id_t` for a parallel region is unique across all parallel regions. A parallel region is assigned an ID when the region is created. A parallel region's ID is passed to callbacks associated with begin/end of the parallel region, as well as callbacks that occur in the context of the parallel region. A parallel region ID can be retrieved on demand by invoking the `ompt_get_parallel_id` function (described in Section **??**). Tools should not assume that `ompt_parallel_id_t` values for adjacent regions are consecutive. The value 0 is reserved to indicate an invalid parallel id.

4.3 Task Region Identifier (Mandatory)

Each OpenMP task region has an associated `ompt_task_id_t` that uniquely identifies the task region. This holds for implicit tasks, including the initial task, as well as for explicit tasks.

```
typedef uint64_t ompt_task_id_t;
```

The `ompt_task_id_t` for a task region is unique across all task regions. A task region is assigned an ID when the region is created. A task region's ID is passed to callbacks associated with begin/end of the task region. A task region's ID can be retrieved on demand by invoking the `ompt_get_task_id` function (described in Section **??**). Tools should not assume that `ompt_task_id_t` values for adjacent task regions are consecutive. The value 0 is reserved to indicate an invalid task id.

An initial task will also have its own unique task region ID.

exit / reenter	reenter = null	reenter = defined
exit = null	case 1) initial task in user code case 2) explicit task that is created but not yet scheduled	task in runtime because of a parallel region or a task creation
exit = defined	non-initial task in (or soon to be in) user code	non-initial task in runtime because of a parallel region or a task creation

Table 1: Meaning of various values for `exit_runtime_frame` and `reenter_runtime_frame`.

4.4 Wait Identifier (Mandatory)

Each thread instance maintains an `ompt_wait_id_t`. When a thread is waiting for something, the thread's wait ID identifies what the thread is awaiting.

```
typedef uint64_t ompt_wait_id_t;
```

For example, when a thread is waiting for a lock, the thread's wait ID identifies the lock. The thread's wait ID is passed to callbacks associated with wait events, and also can be retrieved on demand by invoking the `ompt_get_state` function (described in Section **??**). When a thread is not in a wait state, a thread's wait ID has an undefined value. Value 0 is reserved to indicate an undefined wait ID.

4.5 Pointers to Support Classification of Stack Frames (Mandatory)

Each implicit or explicit task region provides an `ompt_frame_t` data structure which contains pointers to OpenMP runtime procedure frames that appear above and below procedure frames associated with user task code.

```
typedef struct ompt_frame_s {
    void *exit_runtime_frame;    /* next frame is user code     */
    void *reenter_runtime_frame; /* previous frame is user code */
} ompt_frame_t;
```

The structure's lifetime begins when a task region is created and ends when the task region is destroyed. While the value of the structure is preserved over the lifetime of the task, tools should not assume that the address of a structure remains constant over its lifetime. Frame data is passed to some callbacks; it can also be retrieved asynchronously for a task by invoking the `ompt_get_task_frame` function (described in Section **??**) in a signal handler. Frame data contains two components:

`exit_runtime_frame` This value is set once, the first time that a task exits the runtime to begin executing user code. This field points to the stack frame of the runtime procedure that called the user code. This value is NULL until just before the task exits the runtime.

`reenter_runtime_frame` This value is set each time that current task re-enters the runtime to create new (implicit or explicit) tasks. This field points to the stack frame of the runtime procedure called by a task to re-enter the runtime. This value is NULL until just after the task re-enters the runtime.

Table **??** describes the meaning of this structure with various values. In the presence of nested parallelism, a tool may observe a sequence of `ompt_frame_t` records for a thread. Appendix **??** discusses an example that illustrates the use of `ompt_frame_t` records with nested parallelism.

The live range of the `ompt_frame_t` for a given task starts at the `ompt_event_task_begin`, `ompt_event_implicit_task_begin`, or `ompt_event_initial_task_begin` event and lasts until the `ompt_event_task_end`, `ompt_event_implicit_task_end`, or `ompt_event_initial_task_end` event, inclusively.

Advice to tool implementers: A monitoring tool using asynchronous sampling can observe values of `exit_runtime_frame` and `reenter_runtime_frame` at inconvenient times. Tools must be prepared to observe and handle frame exit and reenter values that have not yet been set or reset as the program enters or returns to the runtime.

5 Inquiry Functions for Tools

Inquiry functions retrieve data from the execution environment for the tools. All functions in the inquiry API are marked with `OMPT_API`. These functions should not be global symbols in an OpenMP runtime system implementation to avoid tempting tool developers to call them directly. Section **??** describes how a tool will obtain pointers to these inquiry functions. *All inquiry functions are async signal safe.* Note that it is unsafe to call OpenMP Execution Library Routines within an OMPT callback because doing so may cause deadlock. Specifically, since OpenMP Execution Library Routines are not guaranteed to be async signal safe, they might acquire a lock that may already be held when an OMPT callback is involved.

5.1 Enumerate States Supported by an OpenMP Runtime (Mandatory)

An OpenMP runtime system is allowed to support other states in addition to those described in this document. For instance, a particular runtime system may want to provide more detail about the nature of runtime overhead, e.g., to differentiate between overhead associated with setting up a parallel region and overhead associated with setting up a task. Further, a tool need not report all states defined herein, e.g., if state tracking for a particular state would be too expensive. To enable a tool to identify all states that an OpenMP runtime system implements, OMPT provides the following interface for enumerating all states that a particular runtime system implementation may report.

```
OMPT_API int ompt_enumerate_state(
  ompt_state_t current_state,
  ompt_state_t *next_state,
  const char **next_state_name
);
```

To begin enumerating the states that a runtime system supports, the value `ompt_state_first` should be supplied for `current_state` in the call to `ompt_enumerate_state` that begins the enumeration. The argument `next_state` is a pointer to an `ompt_state_t` that will be set to the code for the next state in the enumeration. The argument `next_state_name` is a pointer to a location that will be filled in with a pointer to the name associated with `next_state`. Subsequent invocations of `ompt_enumerate_state` should pass the code returned in `next_state` by the prior call. Whenever one or more states are left in the enumeration, `ompt_enumerate_state` will return 1. When the last state in the enumeration is passed to `ompt_enumerate_state` as `current_state`, the function will return 0 indicating that the enumeration is complete. An example of how to enumerate the states supported by an OpenMP runtime system is shown below:

```
ompt_state_t state = ompt_state_first;
const char *state_name;
while (ompt_enumerate_state(state, &state, &state_name)) {
  // tool notes that the runtime supports ompt_state_t "state"
  // associated with "state_name"
}
```

5.2 Thread Inquiry (Mandatory)

Function `ompt_get_thread_id` is the inquiry function to determine the thread ID of the current thread.

```
OMPT_API ompt_thread_id_t ompt_get_thread_id();
```

ancestor level	meaning
0	current parallel region
1	parallel region directly enclosing region at ancestor level 0
2	parallel region directly enclosing region at ancestor level 1
...	

Table 2: Meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_id`.

This function returns the value 0 if the thread is unknown to the OpenMP runtime. *This function is async signal safe.*

Function `ompt_get_state` is the inquiry function to determine the state of the current thread.

```
OMPT_API ompt_state_t ompt_get_state(
  ompt_wait_id_t *wait_id
);
```

The function returns the state of the current thread and updates the location specified by `wait_id` with the wait identifier associated with the current state, if any, or zero if the wait ID is undefined. One may pass NULL for `wait_id` if the tool does not want a wait ID returned. *This function is async signal safe.*

Function `ompt_get_idle_frame` is the inquiry function to determine the lowest frame in the current thread's call stack where the thread would await new work.

```
OMPT_API void * ompt_get_idle_frame();
```

We specify the lowest frame where a thread would await work since the thread might call a routine to check for work or a routine that blocks on a condition variable from this frame. The function `ompt_get_idle_frame` returns the value NULL when the current thread has no idle frame in its call stack. Note that this function always returns NULL for an OpenMP initial thread. *This function is async signal safe.*

5.3 Parallel Region Inquiry (Mandatory)

Function `ompt_get_parallel_id` returns the unique ID associated with a parallel region:

```
OMPT_API ompt_parallel_id_t ompt_get_parallel_id(
  int ancestor_level
);
```

Outside a parallel region, `ompt_get_parallel_id` should return 0. If a thread is in the idle state, then `ompt_get_parallel_id` should return 0. In all other cases, the thread should return the ID of the enclosing parallel region, even if the thread is waiting at a barrier.

The function takes an ancestor level as an argument. By specifying different values for ancestor level, one can access information about all enclosing parallel regions. The meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_id` is given in Table **??**.

The function returns the value 0 when requesting higher levels of ancestry than exist. *This function is async signal safe.*

Function `ompt_get_parallel_team_size` returns the number of threads associated with a parallel region:

```
OMPT_API int ompt_get_parallel_team_size(
  int ancestor_level
);
```

This function returns the value -1 when requesting higher levels of ancestry than exist. *This function is async signal safe.*

5.4 Task Region Inquiry (Mandatory)

The OMPT interface defines two inquiry functions that provide information about both implicit and explicit tasks. Both functions `ompt_get_task_id` and `ompt_get_task_frame` specify a target task by a depth. Depth 0 refers to the current task. Information about other tasks in the current execution context may be queried at higher depths. *Both functions are async signal safe.*

The function `ompt_get_task_id` returns an ID for the specified task region.

```
OMPT_API ompt_task_id_t *ompt_get_task_id(
  int depth
);
```

If a tool requests a task ID at a depth deeper than the dynamic nesting of implicit and explicit tasks in the current execution context, `ompt_get_task_id` will return 0—the value reserved to indicate an invalid task.

Function `ompt_get_task_frame` returns an `ompt_frame_t` record that identifies a contiguous interval of frames on the call stack. This interval of stack frames represents activity by the application rather than the OpenMP runtime system.

```
OMPT_API ompt_frame_t *ompt_get_task_frame(
  int depth
);
```

Using return values from `ompt_get_task_frame`, a tool that collects the call stack of a thread can analyze frames in the call stack and identify ones that exist on behalf of the runtime system.[3] This capability enables a tool to map from an implementation-level view back to the source-level view familiar to application developers. Appendix **??** discusses an example that illustrates the use of `ompt_get_task_frame` with multiple threads and nested parallelism.

6 Initializing OMPT Support for Tools

If no tool registers itself with an OpenMP runtime during initialization (as described below in Section **??**), the runtime need not maintain information to support tools and the runtime's behavior is undefined if a tool invokes any API inquiry functions. Section **??** describes tool initialization. Section **??** describes environment variable control over tool initialization.

6.1 Initialization of a Tool

A tool must register itself with an OpenMP runtime system and then specify callbacks for events of interest. Section **??** describes the initializer for a tool. Section **??** describes registration of callbacks for OMPT events.

6.1.1 Initializer (Mandatory)

A tool must register itself with an OpenMP runtime by providing an implementation of the following function:

```
extern "C" {
  int ompt_initialize(ompt_function_lookup_t lookup,
                      const char *runtime_version,
                      unsigned int ompt_version);
}
```

Since only one tool-provided definition of `ompt_initialize` will be seen by an OpenMP runtime, only one tool may register itself. Ordinarily, `ompt_initialize` will be invoked by an OpenMP runtime immediately after the runtime initializes itself.

[3] A frame on the call stack is said to exist on behalf of an OpenMP runtime system if it is a frame for a runtime system routine, or if it belongs to a library function called by a runtime system routine, directly or indirectly.

The function `ompt_initialize` serves two roles. First, if a tool wants to receive notification of OpenMP events (described in Section **??**), the tool's implementation of `ompt_initialize` must register a callback for every event of interest using `ompt_set_callback` (described in Section **??**). Second, the return value of `ompt_initialize` indicates whether or not a tool wants the OpenMP runtime system to maintain thread runtime state information for the tool and invoke any callback functions that the tool may have registered.

A tool-supplied implementation may return 0 or 1. If `ompt_initialize` returns 0, the OpenMP runtime *need not* maintain any state information for OpenMP threads and *will not* perform any callbacks. If a tool-supplied implementation of `ompt_initialize` returns 1, the OpenMP runtime system will maintain state information for each OpenMP thread and will perform any callbacks that have been registered by the tool.

The first argument to `ompt_initialize` is `lookup`—a callback that a tool must use to interrogate the runtime system to obtain pointers to OMPT interface functions. The type signature for `lookup` is:

```
ompt_interface_fn_t lookup(const char *interface_function_name);
```

The `lookup` callback is necessary because when the OpenMP runtime is dynamically loaded by a shared library, the OMPT interface functions provided by the library may be invisible to a preloaded tool. Within a tool, one uses `lookup` to obtain function pointers to each OMPT inquiry function. For example, to obtain a function pointer to `ompt_get_thread_id`, one invokes `lookup` as follows:

```
ompt_interface_fn_t ompt_get_thread_id_ptr = lookup("ompt_get_thread_id");
```

If a named callback is not available in an OpenMP runtime's implementation of OMPT, `lookup` will return NULL.

The second argument to `ompt_initialize` is a version string that unambiguously identifies an OpenMP runtime system implementation. This argument is useful to tool developers trying to debug a statically-linked executable that contains both a tool implementation and an OpenMP runtime system implementation. Knowing exactly what version of an OpenMP runtime system is present in a binary may be helpful when diagnosing a problem, e.g., identifying an old runtime system that may be incompatible with a newer tool.

The third argument `ompt_version` indicates the version of the OMPT interface supported by a runtime system. The version of OMPT described by this document is known as version 1.

After a process fork, if OpenMP is re-initialized in the child process, the OpenMP runtime system in the child process will call `ompt_initialize` under the same conditions as it would for any process.

6.1.2 Callback Registration (Mandatory)

Tools register callbacks to receive notification of various events that occur as an OpenMP program executes. All functions in the registration API are marked with `OMPT_API`. These functions should not be global symbols in an OpenMP runtime system implementation to avoid tempting tool developers to call them directly. Section **??** describes how a tool will obtain pointers to these functions. A tool uses `ompt_set_callback` to register callback functions.

```
OMPT_API int ompt_set_callback(
  ompt_event_t event,
  ompt_callback_t callback
);
```

The function `ompt_set_callback` may only be called within the implementation of `ompt_initialize` provided by a tool, as described in Section **??** The possible return codes for `ompt_set_callback` and their meaning is shown in Table **??**. Registration of supported callbacks may fail if this function is called outside `ompt_initialize`. The `ompt_callback_t` type for a callback does not reflect the actual signature of the callback; OMPT uses this generic type to avoid the need to declare a separate registration function for each actual callback type.

The function `ompt_get_callback` may be called at any time to inspect whether a callback has been registered or not.

return code	meaning
0	callback registration error (e.g., callbacks cannot be registered at this time).
1	event may occur; no callback is possible
2	event will never occur in runtime
3	event may occur; callback invoked when convenient
4	event may occur; callback always invoked when event occurs

Table 3: Meaning of return codes for `ompt_set_callback`.

`OMP_TOOL` value	action
disabled	OMPT is disabled regardless of whether a tool is present or not. The OpenMP runtime is not required to maintain any information about thread state nor support any invocation of the inquiry API.
enabled	`ompt_initialize` is called after the OpenMP runtime initializes itself. If the return value from `ompt_initialize` is 1, the OpenMP runtime must maintain runtime state information for each OpenMP thread, respond to any invocations of the inquiry API, and invoke any registered callbacks when appropriate.

Table 4: OpenMP runtime responses to settings of the `OMP_TOOL` environment variable.

```
OMPT_API int ompt_get_callback(
  ompt_event_t event,
  ompt_callback_t *callback
);
```

If a callback has been registered, `ompt_get_callback` will return 1 and set `callback` to the address of the callback function; otherwise, `ompt_get_callback` will return 0.

6.2 An Environment Variable for Tool Initialization (Mandatory)

The environment variable `OMP_TOOL` is used to control tool initialization. Table **??** describes actions an OpenMP runtime system takes in response to various values of `OMP_TOOL`. When `OMP_TOOL` is not defined, its default value is `enabled`.

An OpenMP runtime will attempt to initialize a tool if `OMP_TOOL` is `enabled`. If the OpenMP runtime calls `ompt_initialize`, but no tool-provided version of `ompt_initialize` is present, a weak version of `ompt_initialize` provided by the OpenMP runtime will return 0. If a tool-provided version of `ompt_initialize` is present, it may return 0 or 1. Only if `ompt_initialize` returns 1 is the OpenMP runtime obligated to perform state tracking and invoke any event callbacks registered by `ompt_initialize`.

If `OMP_TOOL` is set to `disabled`, the OpenMP runtime will not call `ompt_initialize` to attempt tool initialization, maintain any thread state information for tools, or make any tool callbacks.

Behavior for any other values of `OMP_TOOL` is unspecified.

6.3 Implementation Considerations for Tool Initialization

Unless `OMP_TOOL=disabled`, if a tool-supplied implementation of `ompt_initialize` is present in the address space of a process and visible to the OpenMP runtime system, the tool-supplied `ompt_initialize` must be called immediately after the OpenMP runtime system initializes itself.

Whether a tool-supplied implementation of `ompt_initialize` defined as a strong global symbol is visible to an OpenMP runtime system when present in the address space of a process is non-obvious. There are several scenarios to consider. A tool-supplied version of `ompt_initialize` is visible to an OpenMP runtime system if:

- The tool implementation of `ompt_initialize` is statically-linked into an executable. Such an implementation of `ompt_initialize` will be visible to an OpenMP runtime system regardless of whether the runtime is statically linked into the executable or dynamically-linked into a shared library.
- An implementation of `ompt_initialize` is in a tool's shared library, which we denote $\mathcal{L}_T$. Such an implementation of `ompt_initialize` will be visible to an OpenMP runtime system in a library $\mathcal{L}_O$ as long as (a) $\mathcal{L}_O$ is a shared library itself, and (b) $\mathcal{L}_T$ is in the dynamic library search path for $\mathcal{L}_O$ ahead of $\mathcal{L}_O$ itself. $\mathcal{L}_T$ is guaranteed to be on $\mathcal{L}_O$'s dynamic library search path ahead of $\mathcal{L}_O$ *iff*
  - $\mathcal{L}_T$ is pre-loaded by the dynamic linker into the address space of a process before execution begins.[4]
  - $\mathcal{L}_T$ and $\mathcal{L}_O$ are both direct shared library dependences of a load module[5] and $\mathcal{L}_T$ appeared ahead of $\mathcal{L}_O$ when linking the load module.
  - A load module dynamically loads $\mathcal{L}_T$ ahead of a shared library $\mathcal{L}_X$ (because $\mathcal{L}_T$ preceded $\mathcal{L}_X$ when the load module was linked), and $\mathcal{L}_X$ directly or indirectly loads $\mathcal{L}_O$.

The recommended approach for handling initialization in the OpenMP runtime system for a particular target platform depends on the features supported by compiler, linker, and operating system.

Compiler and linker support weak symbols. On systems where the compiler and linker support weak symbols, it is convenient for the OpenMP runtime system to define `ompt_initialize` as a weak global symbol that returns 0. Definition of `ompt_initialize` as a weak global symbol is suitable for use in either a static or dynamic library. If a shared-library implementation of an OpenMP library $\mathcal{L}_O$ defines `ompt_initialize` as a weak global symbol, then a tool library $\mathcal{L}_T$ must be appear on the dynamic library search path ahead of $\mathcal{L}_O$ for the tool version of `ompt_initialize` to be invoked.

Compiler and linker don't support weak symbols. On systems that don't support weak symbols, different implementation strategies are needed for static and dynamic linking.

For a static library implementation of an OpenMP runtime library, the library can provide a stub version of `ompt_initialize` in a separate object file. In this case, the linker will include the OpenMP library's stub implementation of `ompt_initialize` only if no tool supplied version is already present when the OpenMP runtime library is used to resolve undefined symbols.

An OpenMP implementation used as a dynamic library can define `ompt_initialize` as a global symbol. The version in the OpenMP library would be invoked only if no tool-supplied implementation of `ompt_initialize` is statically linked in the executable or a tool library that appears before the OpenMP runtime library in the dynamic library search path during execution.

A Binary rewriter alters a load module that provides an OpenMP runtime system. Regardless of whether a system supports weak symbols or not, one can use a static or dynamic binary rewriting tool to modify an OpenMP runtime system present in an executable or a shared library to invoke a tool-supplied version of a version of `ompt_initialize` rather than the default implementation of `ompt_initialize` present in the OpenMP runtime.

7 Tool Control for Applications (Mandatory)

In OMPT, there is only one application-facing routine: `ompt_control`. An application may call the function `ompt_control` to control tool operation. While tool support for `ompt_control` is optional, the runtime is required to pass a control command to a tool if the tool registered a callback with the `ompt_event_control` event. As an application-facing routine, this function has type signatures for both C and Fortran:

[4] While Linux and some other operating systems support library pre-loading, library pre-loading is not universally available.
[5] A load module is an application binary or a shared library.

```
C:
   void ompt_control(uint64_t command, uint64_t modifier);


Fortran:
   subroutine ompt_control(command, modifier)
   integer*8 command, modifier
```

A classic use case for `ompt_control` is for an application to start and stop data collection by a tool. A tool may allow an application to turn monitoring on and off multiple times during an execution to monitor only code regions of interest. To simplify use in this common case, OMPT defines four values for `command`:

```
1: start or restart monitoring
2: pause monitoring
3: flush tool buffers and continue monitoring
4: permanently turn off monitoring
```

A command code of 1 asks a tool to start or restart monitoring if it is off. If monitoring is already on, this command is idempotent. If monitoring has already been turned off permanently, this command will have no effect. A command code of 2 asks a tool to temporarily turn monitoring off. If monitoring is already off, it is idempotent. A command code of 3 asks a tool to flush any performance data that it has buffered and then continue monitoring. A command code of 4 turns monitoring off permanently; the tool may perform finalization at this point and write all of its outputs.

Other values of command and modifier appropriate for any tool will be tool specific. Tool-specific commands codes must be ≥ 64. Tools must ignore command codes that they are not explicitly designed to handle and implement callbacks for such codes as no-ops.

8 Final Notes

Developers of many trace-based tools would prefer to have `ompt_event_implicit_task_begin` and `ompt_event_implicit_task_end` included in the mandatory events. As we acquire more experience with OMPT implementations, perhaps these events will be added to the set of mandatory events, if they don't add much overhead to OpenMP runtime implementations when OMPT is disabled.

8.1 Future Enhancements

As OpenMP runtime developers acquire more experience with OpenMP 4.0 features, we envision two types of enhancements to OMPT. First, the current definition of OMPT doesn't provide support for tracking or interrogating task dependences. Second, the current definition of OMPT doesn't provide support for tracking data movement and computation for devices. Eventually, we expect to extend OMPT with support to expose information about dependences to tools, track data movement to/from devices, and track computation on devices.

Acknowledgments

The authors would like to acknowledge Scott Parker at Argonne National Laboratory (ANL) for his role in catalyzing new work on an OpenMP tool API as part of ANL's Mira procurement. The design of OMPT builds upon ideas from both the POMP and Collector tool APIs for OpenMP. Bernd Mohr, Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah designed the POMP API. Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin designed the Collector API.

We would also like to recognize other members of the OpenMP tools working group who contributed ideas and feedback that helped shape the design of OMPT: Brian Bliss, Bronis de Supinski, Alex Grund, Kevin Huck, Marty Itzkowitz, Bernd Mohr, Harald Servat, and Michael Wong.

A OMPT Interface Type Definitions

A.1 Runtime States

When OMPT is enabled, an OpenMP runtime system will maintain information about the state of each OpenMP thread. Below we define an enumeration type that specifies the set of runtime states. The purpose of these states is described in Section **??**.

```
typedef enum {
  /* work states (0..15) */
  ompt_state_work_serial           = 0x00, /* working outside parallel   */
  ompt_state_work_parallel         = 0x01, /* working within parallel    */
  ompt_state_work_reduction        = 0x02, /* performing a reduction     */

  /* idle (16..31) */
  ompt_state_idle                  = 0x10, /* waiting for work           */

  /* overhead states (32..63) */
  ompt_state_overhead              = 0x20, /* non-wait overhead          */

  /* barrier wait states (64..79) */
  ompt_state_wait_barrier          = 0x40, /* generic barrier            */
  ompt_state_wait_barrier_implicit = 0x41, /* implicit barrier           */
  ompt_state_wait_barrier_explicit = 0x42, /* explicit barrier           */

 /* task wait states (80..95) */
  ompt_state_wait_taskwait         = 0x50, /* waiting at a taskwait      */
  ompt_state_wait_taskgroup        = 0x51, /* waiting at a taskgroup     */

  /* mutex wait states (96..111) */
  ompt_state_wait_lock             = 0x60, /* waiting for lock           */
  ompt_state_wait_nest_lock        = 0x61, /* waiting for nest lock      */
  ompt_state_wait_critical         = 0x62, /* waiting for critical       */
  ompt_state_wait_atomic           = 0x63, /* waiting for atomic         */
  ompt_state_wait_ordered          = 0x64, /* waiting for ordered        */

  /* misc (112.127) */
  ompt_state_undefined             = 0x70, /* undefined thread state     */
  ompt_state_first                 = 0x71, /* initial enumeration state  */
} ompt_state_t;
```

A.2 Runtime Event Callbacks

When OMPT support for a tool is enabled, OMPT enables a tool to indicate interest in receiving notification about certain OpenMP runtime events by registering callbacks. When those events occur during execution, OMPT will invoke the registered callback in the appropriate thread context. Below we define an enumeration type that specifies the set of event callbacks that may be supported by an OpenMP runtime system. The purpose of these callbacks is described in Section **??**.

```
typedef enum {
  /*--- Mandatory Events ---*/
  ompt_event_parallel_begin           = 1,  /* parallel create         */
  ompt_event_parallel_end             = 2,  /* parallel exit           */

  ompt_event_task_begin               = 3,  /* task create             */
  ompt_event_task_end                 = 4,  /* task destroy            */

  ompt_event_thread_begin             = 5,  /* thread begin            */
  ompt_event_thread_end               = 6,  /* thread end              */

  ompt_event_control                  = 7,  /* support control calls   */

  ompt_event_runtime_shutdown         = 8,  /* runtime shutdown        */

  /*--- Optional Events (blame shifting) ---*/
  ompt_event_idle_begin               = 9,  /* begin idle state        */
  ompt_event_idle_end                 = 10, /* end idle state          */

  ompt_event_wait_barrier_begin       = 11, /* begin wait at barrier   */
  ompt_event_wait_barrier_end         = 12, /* end wait at barrier     */
  ompt_event_wait_taskwait_begin      = 13, /* begin wait at taskwait  */
  ompt_event_wait_taskwait_end        = 14, /* end wait at taskwait    */
  ompt_event_wait_taskgroup_begin     = 15, /* begin wait at taskgroup */
  ompt_event_wait_taskgroup_end       = 16, /* end wait at taskgroup   */

  ompt_event_release_lock             = 17, /* lock release            */
  ompt_event_release_nest_lock_last   = 18, /* last nest lock release  */
  ompt_event_release_critical         = 19, /* critical release        */
  ompt_event_release_atomic           = 20, /* atomic release          */
  ompt_event_release_ordered          = 21, /* ordered release         */

  /*--- Optional Events (synchronous events) --- */
  ompt_event_implicit_task_begin      = 22, /* implicit task create    */
  ompt_event_implicit_task_end        = 23, /* implicit task destroy   */

  ompt_event_initial_task_begin       = 24, /* initial task create     */
  ompt_event_initial_task_end         = 25, /* initial task destroy    */

  ompt_event_task_switch              = 26, /* task switch             */

  ompt_event_loop_begin               = 27, /* task at loop begin      */
  ompt_event_loop_end                 = 28, /* task at loop end        */

  ompt_event_sections_begin           = 29, /* task at section begin   */
  ompt_event_sections_end             = 30, /* task at section end     */
```

```
  ompt_event_single_in_block_begin    = 31, /* task at single begin    */
  ompt_event_single_in_block_end      = 32, /* task at single end      */
  ompt_event_single_others_begin      = 33, /* task at single begin    */
  ompt_event_single_others_end        = 34, /* task at single end      */

  ompt_event_workshare_begin          = 35, /* task at workshare begin */
  ompt_event_workshare_end            = 36, /* task at workshare end   */

  ompt_event_master_begin             = 37, /* task at master begin    */
  ompt_event_master_end               = 38, /* task at master end      */

  ompt_event_barrier_begin            = 39, /* task at barrier begin   */
  ompt_event_barrier_end              = 40, /* task at barrier end     */

  ompt_event_taskwait_begin           = 41, /* task at taskwait begin  */
  ompt_event_taskwait_end             = 42, /* task at task wait end   */

  ompt_event_taskgroup_begin          = 43, /* task at taskgroup begin */
  ompt_event_taskgroup_end            = 44, /* task at taskgroup end   */

  ompt_event_release_nest_lock_prev   = 45, /* prev nest lock release  */

  ompt_event_wait_lock                = 46, /* lock wait               */
  ompt_event_wait_nest_lock           = 47, /* nest lock wait          */
  ompt_event_wait_critical            = 48, /* critical wait           */
  ompt_event_wait_atomic              = 49, /* atomic wait             */
  ompt_event_wait_ordered             = 50, /* ordered wait            */

  ompt_event_acquired_lock            = 51, /* lock acquired           */
  ompt_event_acquired_nest_lock_first = 52, /* 1st nest lock acquired  */
  ompt_event_acquired_nest_lock_next  = 53, /* next nest lock acquired */
  ompt_event_acquired_critical        = 54, /* critical acquired       */
  ompt_event_acquired_atomic          = 55, /* atomic acquired         */
  ompt_event_acquired_ordered         = 56, /* ordered acquired        */

  ompt_event_init_lock                = 57, /* lock init               */
  ompt_event_init_nest_lock           = 58, /* nest lock init          */

  ompt_event_destroy_lock             = 59, /* lock destruction        */
  ompt_event_destroy_nest_lock        = 60, /* nest lock destruction   */

  ompt_event_flush                    = 61  /* after executing flush   */

} ompt_event_t;
```

A.3 Type Signatures for Tool Callbacks

This section describes type signatures for all callbacks that a tool may register to receive from an OpenMP runtime. Section **??** describes OpenMP runtime events and registration of callback functions with these type signatures.

```
/* initialization */
typedef void (*ompt_interface_fn_t)(
  void
  );

typedef ompt_interface_fn_t (*ompt_function_lookup_t)(
  const char *entry_point           /* entry point to look up       */
  );

/* threads */
typedef void (*ompt_thread_callback_t) ( /* for thread              */
  ompt_thread_id_t thread_id        /* ID of thread                 */
  );

typedef enum ompt_thread_type_e {
  ompt_thread_initial = 1,
  ompt_thread_worker  = 2,
  ompt_thread_other   = 3
} ompt_thread_type_t;

typedef void (*ompt_thread_type_callback_t) ( /* for thread         */
  ompt_thread_type_t thread_type,   /* type of thread               */
  ompt_thread_id_t thread_id        /* ID of thread                 */
  );

typedef void (*ompt_wait_callback_t) ( /* for wait                  */
  ompt_wait_id_t wait_id            /* wait ID                      */
  );

/* parallel & workshares */
typedef void (*ompt_parallel_callback_t) ( /* for inside parallel   */
  ompt_parallel_id_t parallel_id,   /* ID of parallel region        */
  ompt_task_id_t  task_id           /* ID of task                   */
  );

typedef void (*ompt_new_workshare_callback_t) ( /* for workshares   */
  ompt_parallel_id_t parallel_id,   /* ID of parallel region        */
  ompt_task_id_t  task_id,          /* ID of task                   */
  void *workshare_function          /* pointer to outlined function */
  );

typedef void (*ompt_new_parallel_callback_t) ( /* for new parallel  */
  ompt_task_id_t parent_task_id,    /* ID of parent task            */
  ompt_frame_t *parent_task_frame,  /* frame data of parent task    */
  ompt_parallel_id_t parallel_id,   /* ID of parallel region        */
  uint32_t requested_team_size,     /* requested number of threads  */
  void *parallel_function           /* pointer to outlined function */
  );
```

```
/* tasks */
typedef void (*ompt_task_callback_t) ( /* for tasks                */
  ompt_task_id_t  task_id           /* ID of task                  */
  );

typedef void (*ompt_task_switch_callback_t) ( /* for task switch    */
  ompt_task_id_t suspended_task_id, /* ID of suspended task        */
  ompt_task_id_t resumed_task_id    /* ID of resumed task          */
  );

typedef void (*ompt_new_task_callback_t) ( /* for new tasks        */
  ompt_task_id_t parent_task_id,    /* ID of parent task           */
  ompt_frame_t *parent_task_frame,  /* frame data for parent task  */
  ompt_task_id_t new_task_id,       /* ID of created task          */
  void *new_task_function           /* pointer to outlined function */
  );

/* program */
typedef void (*ompt_control_callback_t) ( /* for control           */
  uint64_t command,                 /* command of control call     */
  uint64_t modifier                 /* modifier of control call    */
  );

typedef void (*ompt_callback_t)(    /* for shutdown                */
  void
  );
```

Placeholder callback signature. The type `ompt_callback_t` is also a placeholder signature used only by the tool callback registration interface. Only one callback registration function is defined and it expects that the callback supplied will be cast into type `ompt_callback_t`, regardless of its actual type signature. This approach avoids the need for a separate registration routine for each unique tool callback signature.

A.4 OMPT Inquiry and Control API

The functions in this section are not global function symbols in an OpenMP runtime system. These functions can be looked up by name using the `ompt_function_lookup_t` function passed to `ompt_initialize`, as described in Section ?? and Appendix ??.

```
/* callback management */
OMPT_API int ompt_set_callback( /* register a callback for an event                    */
  ompt_event_t event,           /* the event of interest                               */
  ompt_callback_t callback      /* function pointer for the callback                   */
);

OMPT_API int ompt_get_callback( /* return the current callback for an event (if any)  */
  ompt_event_t event,           /* the event of interest                               */
  ompt_callback_t *callback     /* pointer to receive the return value                 */
);

/* state inquiry */
OMPT_API int ompt_enumerate_state( /* extract the set of states supported             */
  ompt_state_t current_state,   /* current state in the enumeration                    */
  ompt_state_t *next_state,     /* next state in the enumeration                       */
  const char **next_state_name  /* string description of next state                    */
);

/* thread inquiry */
OMPT_API ompt_thread_id_t ompt_get_thread_id( /* identify the current thread         */
  void
);

OMPT_API ompt_state_t ompt_get_state( /* get the state for a thread                    */
  ompt_wait_id_t *wait_id        /* for wait states: identify what awaited           */
);

OMPT_API void * ompt_get_idle_frame(  /* identify the idle frame (if any) for a thread */
  void
);

/* parallel region inquiry */
OMPT_API ompt_parallel_id_t ompt_get_parallel_id( /* identify a parallel region     */
  int ancestor_level /*  how many levels the ancestor is removed from the current region */
);

OMPT_API int ompt_get_parallel_team_size( /* query # threads in a parallel region    */
  int ancestor_level /*  how many levels the ancestor is removed from the current region */
);

/* task inquiry */
OMPT_API ompt_task_id_t *ompt_get_task_id(  /* identify a task                       */
  int depth                     /*  how many levels removed from the current task    */
);

OMPT_API ompt_frame_t *ompt_get_task_frame(
  int depth                      /*  how many levels removed from the current task   */
);
```

A.5 Initialization

The function `ompt_initialize` is the only global symbol associated with OMPT. To use OMPT, a tool overlays an implementation of `ompt_initialize` in place of the default one provided by the OMPT implementation. This interface is described in greater detail in Section **??**.

```
extern "C" {
  int ompt_initialize(
    ompt_function_lookup_t lookup,  /* function to look up OMPT API routines by name */
    const char *runtime_version,    /* OpenMP runtime version string                 */
    unsigned int ompt_version       /* integer that identifies the OMPT revision     */
  );
}
```

B Task Frame Management and Inspection

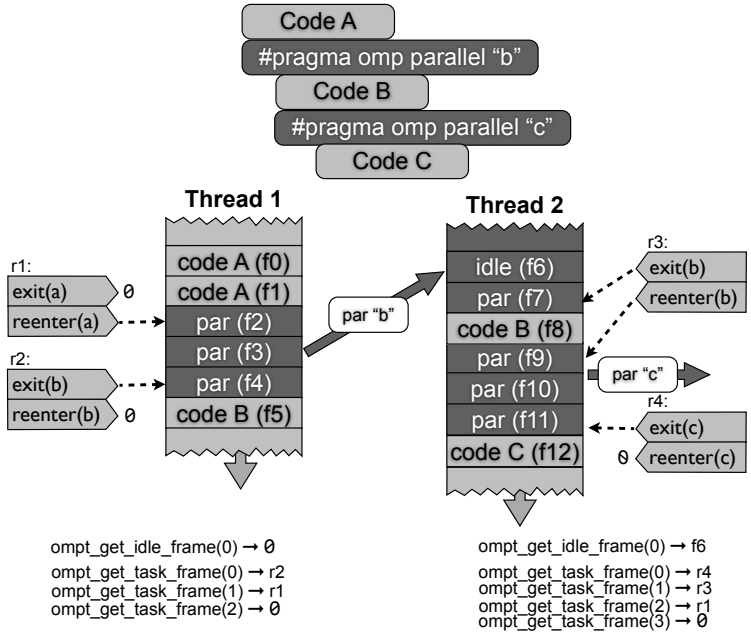

Figure 1: Frame information.

Figure ?? illustrates a program executing a nested parallel region, where code A, B, and C represent, respectively, code associated with an initial task, outer-parallel, and inner-parallel regions. Figure ?? also depicts the stacks of two threads, where each new function call instantiates a new stack frame below the previous frames. When thread 1 encounters the outer-parallel region (parallel "b"), it calls a routine in the OpenMP runtime system to create a new parallel region. The OpenMP runtime sets the `reenter_runtime_frame` field in the `ompt_frame_t` for the initial task executing code A to frame f2, the runtime routine called by frame f1 in the initial task. The `ompt_frame_t` for the initial task is labeled `r1` in Figure ??. In this figure, three consecutive runtime system frames (labeled "par" with frame identifiers f2–f4) are on the stack. Before starting the implicit task for parallel region "b" in thread 1, the runtime sets the `exit_runtime_frame` in the implicit task's `ompt_frame_t` (labeled `r2`) to f4. Execution of application code for parallel region "b" begins on thread 1 when the runtime system invokes application code B (frame f5) from frame f4. Since thread 1 is an OpenMP initial thread, a call to `ompt_get_idle_frame` on this thread will always return NULL.

Let us focus now on thread 2, an OpenMP thread. Figure ?? shows this worker executing work for the outer-parallel region "b." On the OpenMP thread's stack is a runtime frame labeled "idle," where the OpenMP thread waits for work. At any time after the idle frame is on thread 2's stack, a call to `ompt_get_idle_frame` by thread 2 will return frame f6. When work becomes available, the runtime system invokes a function to dispatch it. While dispatching parallel work might involve a chain of several calls, here we assume that the length of this chain is 1 (frame f7). Before thread 2 exits the runtime to execute an implicit task for parallel region "b," the runtime sets the `exit_runtime_frame` field of the implicit task's `ompt_frame_t` (labeled `r3`) to frame f7. When thread 2 later encounters the inner-parallel region "c," execution returns to the runtime and the runtime fills in the `reenter_runtime_frame` field of the current task's `ompt_frame_t` (labeled `r3`) to frame f9. Before the task for parallel region "c" is invoked on thread 2, the runtime system sets the `reenter_runtime_frame` field of the `ompt_frame_t` (labeled `r4`) for the implicit task for "c" to frame f11. Execution of application code for parallel region "c" begins on thread 2 when the runtime system invokes application code C (frame f12) from frame f11.

Below the stack for each thread in Figure ?? is set of `ompt_get_task_frame` inquiries that are assumed to be made on each thread for the stack state shown. Each call indicates an ancestor level with an argument

and shows the ID of the `ompt_frame_t` record returned. Note that thread 2 has task frame information for three levels of tasks, whereas thread 1 has only two.