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

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

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

Revised February 3, 2016

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 [3] and the Sun/Oracle Collector API [1, 2]. 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* [4, 5]—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.

[*]IBM T.J. Watson Research Center
[†]Rice University
[‡]Lawrence Livermore National Laboratory
[§]Oracle
[¶]Intel
[‖]RWTH Aachen University
[**]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.

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 8.

In some cases, the OMPT API may enable a tool to infer details and observe performance implications about the implementation chosen by an OpenMP compiler and runtime. An OpenMP implementation may differ from the abstract execution model described by the OpenMP standard. The ability of tools using OMPT to observe such differences does not affect the language implementation's ability to optimize using the "as if" rule described in the OpenMP standard.

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.
  - 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.
  - The OpenMP runtime 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 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 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 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.
- 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 2 describes the state information maintained by the OpenMP runtime on behalf of OMPT for use by tools. Section 3 describes the OMPT callbacks to notify a tool of various OpenMP runtime events during an execution. Section 4 describes the data structures used by the OMPT interface. Section 5 describes the runtime system inquiry operations supported by OMPT for the benefit of tools. Section 7 describes the OMPT API operations for tool initialization. Section 8 describes the tool control interface available to applications. Section 9 concludes with a few notes about potential future enhancements. Appendix A provides a definition of the complete OMPT interface in C. Appendix B 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. Appendix C outlines some considerations that impact the design of the interface for tool registration.

2 Runtime States

To enable a tool to understand what an OpenMP thread is doing, when a tool registers itself with an OpenMP runtime, 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 A.1 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.

Some states must be supported by any compliant implementation, e.g., those indicating that a thread is executing parallel or serial work. In other cases, alternatives exist. For instance, one may use a single state to represent all waiting at barriers or use a pair of states to differentiate between waiting at implicit and explicit barriers. For some states, OpenMP runtimes have some flexibility about whether to report the state early or late. For example, consider when a thread acquires a lock. One compliant runtime may transition a thread's state to `ompt_state_wait_lock` 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.

$\boxed{\text{Idle State}}$

`ompt_state_idle`

The thread is idle, waiting for work.

Work States

`ompt_state_work_serial`

The thread is executing code outside all parallel regions.

`ompt_state_work_parallel`

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

`ompt_state_work_reduction`

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`

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`

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`

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.

Target Wait States

`ompt_state_wait_target`

The thread is waiting at a target construct. A compliant implementation will have a thread enter this state when the thread begins to wait for a target region to complete.

`ompt_state_wait_target_data`

The thread is waiting at a target data construct. A compliant implementation will have a thread enter this state when the thread begins to wait for a target data construct to complete. A compliant runtime implementation may report `ompt_state_wait_target` for target data constructs.

`ompt_state_wait_target_update`

The thread is waiting at a target update construct. A compliant implementation will have a thread enter this state when the thread begins to wait for a target update construct to complete. A compliant runtime implementation may report `ompt_state_wait_target` for target update constructs.

Task Wait States

`ompt_state_wait_taskwait`

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`

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`

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

Before a thread enters this state, the OpenMP runtime will update the thread's `ompt_wait_id_t` field to identify the lock being awaited.

`ompt_state_wait_critical`

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 will update the thread's `ompt_wait_id_t` field to identify the construct's associated *name* or synchronization variable.

`ompt_state_wait_atomic`

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 will update the thread's `ompt_wait_id_t` field to identify the program variable or synchronization variable associated with the atomic construct.

`ompt_state_wait_ordered`

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 will update the thread's `ompt_wait_id_t` field to identify the synchronization variable associated with the ordered construct.

Overhead State

`ompt_state_overhead`

A thread may be reported as being in the overhead state at any point while executing within an OpenMP runtime, 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`

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`

This state is a placeholder exclusively reserved for use by the OMPT runtime call `ompt_enumerate_state` (see Section 5.1), 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 A.2 defines this type. A tool need not register a callback for any particular event. All callbacks are synchronous and will run to completion before another callback will occur on the same thread. In most cases, an OpenMP runtime 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, for certain begin/end event pairs a runtime may assume that if one event of the pair has a callback registered, the other will have a callback registered 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 A.4 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 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 runtime, or in the context of a new thread created by the runtime 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. This callback has type signature `ompt_thread_callback_t`. The callback argument `thread_type` indicates the type of the thread: initial, worker, or other.

`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 worker thread; it is optional for other types of threads. This callback has type signature `ompt_thread_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_parallel_begin_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_end_callback_t`.

Note to implementers: For a degenerate parallel region executed by a single thread, e.g., a nested region encountered when nested parallelism is disabled or at a nesting depth greater than the maximum number of nested active parallel regions supported on a device, it is implementation dependent whether or not an OpenMP runtime will perform `ompt_event_parallel_begin` and `ompt_event_parallel_end` callbacks.

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_task_begin_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`.

`ompt_event_target_task_begin`

The OpenMP runtime invokes this callback after a task encounters a target, target enter data, or target exit data, target update construct but before the target task executes. The callback executes in the context of the task that encountered the target construct. This callback has type signature `ompt_target_task_begin_callback_t`.

`ompt_event_target_task_end`

The OpenMP runtime invokes this callback after a target task completes on both the host and the device. This callback has type signature `ompt_target_task_end_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 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 (described in Section 7.2) 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. The OpenMP runtime 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] Time a thread spends waiting for work can be blamed on active tasks on other threads that aren't shedding enough parallelism to keep all threads busy. Time a task spends waiting for other tasks to arrive or complete in barrier, taskwait, or taskgroup regions can be blamed on tasks late to arrive or complete. The time a task t spends waiting for mutual exclusion can be blamed on any task holding the mutex while t waits. Since waiting indicates the absence of any activity, a thread will not receive any event notification between begin and end notifications for waiting.

`ompt_event_idle`

This callback has type signature `ompt_scoped_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` when a thread begins to idle outside a parallel region. It again invokes this callback with `endpoint=ompt_scope_end` when the thread finishes idling outside a parallel region. The callback executes in the environment of the idling thread.

`ompt_event_sync_region_wait`

This callback has type signature `ompt_scoped_sync_region_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` when a task starts waiting in a barrier region, taskwait region, or taskgroup region. It again invokes this callback with the `endpoint=ompt_scope_end` when the task stops waiting in the region. The argument `kind` indicates the kind of region causing the wait. One region may generate multiple pairs of begin/end callbacks if another task is scheduled on the thread while the task awaiting completion of the region is stalled. The `codeptr_ra` callback argument may be NULL. This callback executes in the context of the task that encountered the barrier, taskwait, or taskgroup construct.

`ompt_event_mutex_release`

The OpenMP runtime invokes this callback after a task releases a lock, performs the outermost release of a nest lock, or exits a critical, ordered, or atomic region. This callback has type signature `ompt_mutex_release_callback_t`. The argument `kind` indicates the kind of release. In some runtime implementations, it may be inconvenient to distinguish the kind of mutex (lock, nest lock, critical region, or atomic region) being released. If so, the runtime may simply report `kind=ompt_mutex`. If there is a matching `ompt_event_mutex_acquire` callback, it should report the same `kind` value.

[2]The utility of blame shifting has previously been demonstrated for attributing the cost of waiting to steal work in a work-stealing runtime [4] or waiting to acquire a lock [5].

The `wait_id` parameter identifies the lock or internal runtime variable associated with critical region, atomic region, or ordered section released. This callback executes in the context of the task that performed the release.

If an atomic region is implemented using a hardware instruction, then an OpenMP runtime may choose never to report a release for the atomic region. However, if an atomic region is implemented using any mechanism that involves a software protocol that spin waits for a lock or retries hardware primitives that can fail, then an OpenMP runtime developer should consider reporting this event to so a task can accept blame for any spin waiting or retries that occurs while the task has exclusive access to the atomic region. Examples of hardware primitives that could fail with explicit retries in software include transactional instructions, load-linked/store-conditional, and compare-and-swap.

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`

This callback has type signature `ompt_scoped_parallel_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` after an implicit task is fully initialized but before the task begins to work. It again invokes this callback with the `endpoint=ompt_scope_end` after the implicit task executes its closing synchronization barrier but before the task is destroyed. This callback executes in the context of the implicit task.

`ompt_event_initial_task`

This callback has type signature `ompt_scoped_task_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` after an initial task is fully initialized but before it begins to work. It again invokes this callback with the `endpoint=ompt_scope_end` after an initial task ends but before the task is destroyed. This callback executes in the context of the initial task.

`ompt_event_task_switch`

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

Task Dependence Tracking

`ompt_event_task_dependences`

The OpenMP runtime invokes this callback after an explicit task is created, but before the task begins execution to announce its dependences with respect to data objects. This callback has type signature `ompt_task_dependences_callback_t`.

`ompt_event_task_dependence_pair`

The OpenMP runtime invokes this callback to report a dependence between a producer (`first_task_id`) and a consumer (`second_task_id`) that blocked the execution of the consumer. This callback will occur before the successor task is notified that the dependence is satisfied. This may happen early or late. Note: this callback will be invoked only to report blocking dependences between sibling tasks whose lifetimes overlap. No callback will occur if a producer task has finished before a consumer task is created. This callback has type signature `ompt_task_pair_callback_t`.

Lock Creation and Destruction

`ompt_event_init_lock`

The OpenMP runtime invokes this callback just after a task initializes a lock or nest lock. This callback executes in the context of the task that called `omp_init_lock` or `omp_init_nest_lock`. This callback has type signature `ompt_lock_callback_t`. The callback's `wait_id` parameter identifies the lock. The `hint` parameter is the lock hint value passed to the initialization request. The `type` parameter is a small integer that indicates the lock implementation assigned by the OpenMP runtime. One can determine the mapping between values of `type` and the names of the lock implementations they represent by using `ompt_enumerate_lock_types`.

`ompt_event_destroy_lock`

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

Worksharing

`ompt_event_worksharing_begin`

The OpenMP runtime invokes this callback after a task encounters a worksharing construct but before the task executes its first unit of work for the worksharing construct. This callback executes in the context of the task that encountered the construct. This callback has type signature `ompt_worksharing_begin_callback_t`. The `wstype` callback argument indicates whether the worksharing construct is a loop, sections, single, or workshare. The `codeptr_ra` callback argument contains the return address of the call to the runtime routine, which relates the operation to the user program.

`ompt_event_worksharing_end`

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

Master Blocks

`ompt_event_master_begin`

The OpenMP runtime 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_sync_callback_t`.

`ompt_event_master_end`

The OpenMP runtime 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`.

Target Devices

`ompt_event_target_data_begin`

The OpenMP runtime invokes this callback after a task encounters a target data construct but before the new data environment is created. The callback executes in the context of the task that encountered the target data construct. This callback has type signature `ompt_target_data_begin_callback_t`. Arguments to the callback include the ID of the encountering task, the ID of the target device, and the return address of the call to the runtime routine performing the target data operation, which relates the operation to the user program.

`ompt_event_target_data_end`

The OpenMP runtime invokes this callback when the task that encountered the target data region is done with the target data region. The callback executes in the context of the task that encountered the target data construct. This callback has type signature `ompt_task_callback_t`.

`ompt_event_target_data_map_begin`

The OpenMP runtime invokes this callback just before a variable is mapped to or from the device data environment. The callback executes in the context of the encountering task, which executes the region. This callback has type signature `ompt_target_data_map_begin_callback_t`.

`ompt_event_target_data_map_end`

The OpenMP runtime invokes this callback immediately after a variable is mapped to or from the device data environment. Since the map operation may execute asynchronously on the device, the map end callback on the host may occur before the map operation is complete. The callback executes in the context of the the encountering task, which executes the region. This callback has type signature `ompt_target_data_map_end_callback_t`. The `map_id` passed to the callback is the same as the one provided to the task that initiated the mapping.

Barrier, Taskwait, and Taskgroup

`ompt_event_sync_region`

This callback has type signature `ompt_scoped_sync_region_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` before a task begins execution of a barrier region, taskwait region, or taskgroup region. It again invokes this callback with `endpoint=ompt_scope_end` before the task exits the synchronization region. The argument `kind` indicates the kind of synchronization region: barrier, taskwait, or taskgroup. The `codeptr_ra` callback argument, which represents the return address of the call to the OpenMP API routine, may be NULL when `endpoint=ompt_scope_end`. This callback executes in the context of the implicit task that encountered the synchronization construct.

Locks

`ompt_event_mutex_acquire`

This callback has type signature `ompt_scoped_mutex_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` when a task invokes `omp_set_lock` to acquire a lock or invokes `omp_set_nest_lock` to acquire a nest lock not already owned. It again invokes this callback with `endpoint=ompt_scope_end` just after the task acquires the lock. This callback executes in the context of the task that called `omp_set_lock` or `omp_set_nest_lock`. The callback argument `wait_id` identifies the lock or nest lock

Nest Locks

`ompt_event_nested_lock_acquire`

This callback has type signature `ompt_scoped_nested_lock_callback_t`. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` if a task begins to acquire a nest lock that is already owned by a task. It again invokes this callback with `endpoint=ompt_scope_end` just after a task completes the nested acquisition. This callback executes in the context of the task that called `omp_set_nest_lock`; its callback argument `wait_id` identifies the nest lock.

`ompt_event_nested_lock_release` The OpenMP runtime invokes this callback after a task releases a nest lock that is still owned by this task after the release. 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`.

Critical Sections

`ompt_event_wait_critical`

The OpenMP runtime 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. This callback has type signature `ompt_lock_callback_t`. The callback's `wait_id` parameter identifies the lock used to implement the critical construct. The `lock_hint` parameter is the lock hint value specified in the critical construct. The `lock_type` parameter is a small integer that indicates the kind of lock assigned by the OpenMP runtime. One can determine the mapping between `lock_type` integers and the names of the lock types they represent by using `ompt_enumerate_lock_types`.

`ompt_event_acquired_critical`

The OpenMP runtime 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 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 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 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_atomic_callback_t`.

`ompt_event_acquired_atomic`

The OpenMP runtime 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 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

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 on a device. 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 5.4). 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

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 on a device. 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 5.5). 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

Each OpenMP task region has an associated `ompt_task_id_t` that uniquely identifies the task region on a device. 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_info` function (described in Section 5.6). 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.

4.4 Wait Identifier

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 5.4). 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.

exit / reenter	reenter = null	reenter = defined
exit = null	case 1) initial task in user code case 2) task that is created but not yet scheduled	task entered the runtime to create an implicit, explicit, or target task
exit = defined	non-initial task in (or soon to be in) user code	non-initial task entered the runtime and created an implicit, explicit, or target task

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

4.5 Pointers to Support Classification of Stack Frames

When executing an OpenMP program, at times, procedure frames from the runtime system appear on the call stack between procedure frames of user code. To enable a tool to classify procedure frames on the call stack as belonging to the user program or the OpenMP runtime in such cases, the runtime system maintains an instance of an `ompt_frame_t` data structure for each (possibly degenerate) task. A task is considered degenerate if a call to the OpenMP runtime to create a task region does not create a new task. One case where a degenerate task can arise is when a parallel construct is encountered while executing in a parallel region and nested parallelism is not enabled. Even a degenerate task region may add runtime frames to the call stack before invoking user code for the degenerate task and thus require an `ompt_frame_t` data structure. To simplify the discussion below, we omit the qualifier "possibly degenerate" each time we use the terms *task* or *task region*.

Each implicit, explicit, or target task region provides an `ompt_frame_t` data structure that contains a pair of pointers. One pointer points to the procedure frame that appears immediately below the bottommost procedure frame of user code for the task. If the task invokes the runtime system to attempt creation of a new task region, the other pointer will point to the procedure frame of user code that invokes the runtime system.

```
typedef struct ompt_frame_s {
    void *exit_runtime_frame;    /* next frame is user code     */
    void *reenter_runtime_frame; /* user frame that reenters the runtime  */
} 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_info` function (described in Section 5.6) in a signal handler. Frame data contains two components:

`exit_runtime_frame` This value is set once, before the runtime begins executing user code for a new implicit, explicit, or target task. This field points to the frame pointer of the procedure frame that invoked the user code for the task region. In a runtime where the master thread for a parallel region invokes user code directly (e.g. `libgomp`), this may point to a frame of user code for the enclosing task. This value is NULL until just before the task invokes the user code.

`reenter_runtime_frame` This value is set each time the current task re-enters the runtime to create a new implicit, explicit, or target task region. This field points to the frame pointer for a user function that invokes the runtime to create the task region. This value is set upon entering the runtime and cleared when the task region ends.

Table 1 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 B discusses an example that illustrates the use of `ompt_frame_t` records with nested parallelism.

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 implementation to avoid tempting tool developers to call them directly. Section 7.2 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 Environment 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

The OpenMP runtime 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 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 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 Enumerate Lock Implementations

The OpenMP runtime will support one or more types of locks. When a lock is initialized, the callback for lock initialization (of type `ompt_lock_callback_t`) provides a small integer to identify the type of the lock being initialized. To understand the type of lock associated with each such small integer value, OMPT provides the following interface for enumerating all lock types that a particular runtime system implements.

```
OMPT_API int ompt_enumerate_lock_types(
  ompt_lock_impl_t current_type,
  ompt_lock_impl_t *next_type,
  const char **next_type_name
);
```

To begin enumerating the lock implementations that a runtime system supports, the value `ompt_lock_type_first` should be supplied for `current_type` in the call to `ompt_enumerate_lock_types` that begins the enumeration. The argument `next_type` is a pointer to an `ompt_lock_impl_t` that will be set to the code for the next implementation in the enumeration. The argument `next_type_name` is a pointer to a location that will be filled in with a pointer to the name of the lock implementation associated with `next_type`. Subsequent invocations of `ompt_enumerate_lock_types` should pass the code returned in `next_type` by the prior call. Whenever one or more lock implementations are left in the enumeration, `ompt_enumerate_lock_types` will return 1. When the last lock implementation type in the enumeration is passed to `ompt_enumerate_lock_types` as `current_type`, the function will return 0 indicating that the enumeration is complete. An example of how to enumerate the types of lock implementations supported by an OpenMP runtime is shown below:

```
ompt_lock_type_t type = ompt_lock_impl_first;
const char *impl_name;
while (ompt_enumerate_lock_types(impl, &impl, &impl_name)) {
  // tool notes that the runtime supports ompt_lock_impl_t "impl"
  // associated with "impl_name"
}
```

5.3 Enumerate Atomic Implementations

The OpenMP runtime may implement atomic operations in multiple ways. Whenever possible, implementations should use hardware support for atomic instructions. When atomic instructions that cannot fail are used, we don't expect them to be observable by OMPT. Of interest to OMPT are implementations atomic operations that may cause significant delays by spinning or blocking in software. This can occur when atomics are implemented using locks or mechanisms that may repeatedly retry, e.g., load-linked/store-conditional or transactional memory. To gain insight into why particular atomic operations may perform poorly, OMPT describes waiting for atomics by providing an `atomic_type`—a small integer that identifies the nature of an atomic implementation.

To understand the implementation strategy associated with each such small integer value of an atomic type, OMPT provides the following interface for enumerating all implementations of atomic types that may cause waiting in software.

```
OMPT_API int ompt_enumerate_atomic_type(
  ompt_atomic_type_t current_type,
  ompt_atomic_type_t *next_type,
  const char **next_type_implementation
);
```

To begin enumerating a runtime's implementations of atomic types that may cause waiting, the value `ompt_atomic_type_first` should be supplied for `current_type` in the call to `ompt_enumerate_atomic_type` that begins the enumeration. The argument `next_type` is a pointer to an `ompt_atomic_type_t` that will be set to the code for the next type in the enumeration. The argument `next_type_implementation` is a pointer to a location that will be filled in with a pointer to a description of the implementation of the atomic type associated with `next_type`. Subsequent invocations of `ompt_enumerate_atomic_type` should pass the code returned in `next_type` by the prior call. Whenever one or more atomic types are left in the enumeration, `ompt_enumerate_atomic_type` will return 1. When the last atomic type in the enumeration is passed to `ompt_enumerate_atomic_type` as `current_type`, the function will return 0 indicating that the enumeration is complete. An example of how to enumerate the atomic types supported by an OpenMP runtime is shown below:

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`.

```
ompt_atomic_type_t type = ompt_atomic_type_first;
const char *type_implementation;
while (ompt_enumerate_atomic_type(type, &type, &type_implementation)) {
  // tool notes that the runtime supports ompt_atomic_type_t "type"
  // associated with "type_implementation"
}
```

5.4 Thread Inquiry

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(void);
```

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.*

5.5 Parallel Region Inquiry

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 2.

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.*

depth	meaning
0	current task
1	task below
2	parent of task at ancestor level 1
...	

Table 3: Meaning of different values for the `depth` argument to `ompt_get_task_info`.

5.6 Task Region Inquiry

An inquiry function provides information about implicit, explicit, target, and degenerate tasks:

```
OMPT_API ompt_bool ompt_get_task_info(
  int ancestor_level,
  ompt_task_type_t *type,
  ompt_task_id_t *task_id,
  ompt_frame_t **task_frame,
  ompt_parallel_id_t *par_id
);
```

Function `ompt_get_task_info` identifies a task by an ancestor level. Ancestor level 0 refers to the current task. Information about other tasks in the current execution context may be queried at higher ancestor levels. Function `ompt_get_task_info` returns an a boolean, which indicates whether task information is available at the requested ancestor level. *This function is async signal safe.*

If presented with a non-NULL `type` pointer, `ompt_get_task_info` will return information about the type of the task at the specified ancestor level. A task may be of serial, implicit, explicit, target, or degenerate type. If presented with a non-NULL `task_id` pointer, `ompt_get_task_info` will return the ID of the task at the specified ancestor level. If the task at the specified ancestor level is a degenerate task, the value for `task_id` will be 0. If presented with a non-NULL `task_frame` pointer, `ompt_get_task_info` will return the `ompt_frame_t` object for the task at the specified ancestor level. Finally, if presented with a non-NULL `par_id` pointer, `ompt_get_task_info` will return the identity of the parallel region to which the task at the specified ancestor level belongs. A degenerate task will be associated with the enclosing parallel region. If the task is outside any parallel region, the value returned in `par_id` will be 0.

Using return values from `ompt_frame_t` objects returned by calls to `ompt_get_task_info`, 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 B discusses an example that illustrates the use of `ompt_frame_t` objects with multiple threads and nested parallelism.

5.7 Target Device Inquiry

Function `ompt_get_num_devices` returns the number of visible devices:

```
OMPT_API int ompt_get_num_devices(void);
```

This inquiry function is only supported on the host. If the inquiry function is invoked by a thread not executing in the scope of a `target`, `target data`, or `target update` construct, the return value is undefined. *This function is async signal safe.*

Function `ompt_get_device_info`, whose type signature is shown below, is used to acquire information about the attached device with index `device_id`.

```
OMPT_API int ompt_target_get_device_info(
  int device_id,
```

[3]A frame on the call stack is said to exist on behalf of an OpenMP runtime 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.

```
    const char **type,
    ompt_target_device_t **device,
    ompt_function_lookup_t *lookup,
    const char *documentation
  );
```

If `device_id` refers to a valid device, the function will return 0 indicating success; otherwise, it will return a non-zero value and the values of its return parameters are undefined. If a non-NULL pointer is passed for `type`, the runtime will set `*type` to point to a character string that identifies at a minimum the type of the device. It might also indicate the software stack it is running and perhaps even the version number of one or more components in that stack. An example string could be "NVIDIA Tesla M2050, compute capability 2.0, CUDA 5.5." A tool can use such a type string to determine if it has any special knowledge about hardware and software of such a device. A non-NULL pointer must be provided for `device`. The runtime will set `*device` to point to an opaque object that represents the target device instance. The device pointer returned will need to be supplied as an argument of calls to device-specific functions in the target interface to identify the device being addressed.

A tool must pass a non-NULL pointer for `lookup`. The runtime will set the value of `*lookup` to point to a function that can be used to look up device-specific API functions. The `lookup` function for a device will enable a tool to look up all functions marked `OMPT_TARG_API`. If a named function is not available in an OpenMP runtime's implementation of OMPT, lookup will return NULL. Documentation for the names and type signatures of any additional device-specific API functions available through `lookup` should be provided in the form of a single character string `documentation` that has embedded newlines to keep the text to 80 characters per line. Ideally, the documentation string should include not only the type signature but also necessary descriptive text for how to use the device-specific API or pointers to external documentation.

Function `ompt_target_get_device_id` returns the device identifier for the active target device:

```
  OMPT_API int ompt_target_get_device_id(void);
```

This inquiry function is only supported on the host. If the inquiry function is invoked by a thread not executing in the scope of a `target`, `target data`, or `target update` construct, then it will return a value of -1. *This function is async signal safe.*

Host and target devices are typically distinct and run independently. If a host and target device are different hardware components, they may use different clock generators. For this reason, there may be no common time base for ordering host-side and device-side events. Function `ompt_target_get_time`, with the type signature below returns the current time stamp on the specified target device:

```
  OMPT_TARG_API ompt_target_time_t ompt_target_get_time(
    ompt_target_device_t *device /* target device handle */
  );
```

This inquiry function can be used to acquire information that can be used to align time stamps from the target device with time stamps from the host or other devices.

The function `ompt_target_translate_time` is used to translate a device time to a host time. The `double` result for the host time has the same meaning as the `double` returned from `omp_get_wtime`.

```
  OMPT_TARG_API double ompt_target_translate_time(
    ompt_target_device_t *device, /* target device handle */
    ompt_target_time_t time
  );
```

Advice to tool implementers: The accuracy of time translations will only get worse the longer they are delayed due to changes in clock speed (e.g., due to thermal issues). Prompt translation of device times to host times is recommended.

6 Target Device Tracing (Optional)

Target devices typically operate asynchronously with respect to a host. It may not be practical or possible to make event callbacks on a target device. These characteristics motivate a design of a performance monitoring

return code	meaning
0	error
1	event may occur but no tracing is possible
2	event will never occur in runtime
3	event may occur and will be traced when convenient
4	event may occur and will always be traced if event occurs

Table 4: Meaning of return codes for `ompt_trace_set_ompt`.

interface for target devices where:

- a target device may execute asynchronously from the host,
- the target device records events that occur during its execution in a trace buffer,
- when a trace buffer fills on a device (or it is otherwise useful to flush the buffer), the device provides it to a tool on the host, by invoking a tool-supplied callback function to process and empty the buffer,
- when the target device needs a new trace buffer, it invokes a tool-supplied callback function to request a new buffer,

Section 6.1 describes how to enable or disable tracing on a target device for specific OMPT events. Section 6.2 describes how to enable or disable tracing on a target device for specific native events. Section 6.3 describes how to start and stop event tracing on a device.

Some functions in the target device tracing control API described in this section are marked with `OMPT_TARG_API`. These represent function pointers that should be obtained from a target device by invoking the `lookup` function (provided by the target as a return value to function `ompt_target_get_device_info`) and passing it the name of the function of interest.

6.1 Enabling and Disabling Tracing for OMPT Record Types

A tool uses function `ompt_target_set_trace_ompt` to enable or disable the recording of trace records for one or more types of OMPT events:

```
OMPT_TARG_API int ompt_target_set_trace_ompt(
  ompt_target_device_t *device,      /* target device handle      */
  ompt_bool enable,                  /* enable or disable         */
  ompt_event_t etype                 /* an event type             */
);
```

The argument `device` is a handle for the target device whose performance monitoring may be altered by invoking this function. The boolean `enable` indicates whether recording of events of type `rtype` may be enabled or disabled by this invocation. Actual record types are specified using positive numbers; an `rtype` of 0 indicates that all record types will be enabled or disabled. Table 4 shows the possible return codes for `ompt_target_set_trace_ompt`.

6.2 Enabling and Disabling Tracing for Native Record Types

A tool uses function `ompt_target_set_trace_native` to enable or disable the recording of native trace records for a device. This interface is designed for use by a tool with no knowledge about an attached device. If a tool knows how to program a particular attached device, it may opt to invoke native control functions direction using pointers obtained through the `lookup` function associated with the device and described in the `documentation` string that is returned by `ompt_target_get_device_info`.

```
OMPT_TARG_API int ompt_target_set_trace_native(
  ompt_target_device_t *device,      /* target device handle            */
  ompt_bool enable,                /* enable or disable              */
  uint32_t  flags                  /* event classes to monitor       */
);
```

The argument `device` is a handle for the target device whose performance monitoring may be altered by invoking this function. The boolean `enable` indicates whether recording of events may be enabled or disabled by this invocation. The kinds of native device monitoring to enable or disable are specified by `flags`. Each kind of monitoring is specified by a flag bit. Flags can be composed by using logical `or` to combine enumeration values from type `ompt_native_mon_flags_t`.

6.3 Start and Stop Recording Traces

To start recording, a tool needs to register a *buffer request* callback that will supply a device with a buffer to deposit events and a *buffer complete* callback that will be invoked by the OpenMP runtime to empty a buffer containing event records. A device's offloading runtime library is responsible for invoking these callbacks on a thread that is not an OpenMP master or worker. *The buffer request and completion callbacks are not required to be async-signal safe.*

The *buffer request* callback has the following type signature:

```
typedef void (*ompt_target_buffer_request_callback_t) (
  int device_id,
  ompt_target_buffer_t **buffer,
  size_t *bytes
);
```

As necessary, the OpenMP runtime will asynchronously invoke `ompt_target_buffer_request_callback_t` to request a buffer to store event records for device `device_id`. A tool should set `*buffer` to point to a buffer where device events may be recorded and `*bytes` to the length of that buffer. A buffer request callback may set `*bytes` to 0 if it does not want to provide a buffer for any reason. If a callback sets `*bytes` to 0, further recording of events for the device will be disabled until the next invocation of `ompt_target_start_trace`. This will cause the target device to drop future trace records until recording is restarted.

The *buffer complete* callback has the following type signature:

```
typedef void (*ompt_target_buffer_complete_callback_t) (
  int device_id,
  const ompt_target_buffer_t *buffer,
  size_t bytes,
  ompt_target_buffer_cursor_t begin,
  ompt_target_buffer_cursor_t end
);
```

A target device triggers a call to `ompt_target_buffer_complete_callback_t` when no further records will be recorded in an event buffer and all records written to the buffer are valid. The argument `device_id` indicates the device whose events the buffer contains. The argument `buffer` is the address of a buffer previously allocated by the *buffer request* callback. The argument `bytes` indicates the full size of the buffer. The arguments `begin` and `end`, respectively, are opaque cursors that indicates the position at the beginning of the first record in the buffer and the position after the last record in the buffer. If `begin` is equal to `end`, the buffer is empty.

Under normal operating conditions, every event buffer provided to a device by a *buffer request* callback will receive a *buffer complete* callback before the OpenMP runtime shuts down. If an exceptional condition terminates execution of an OpenMP program, the OpenMP runtime may choose to not provide a *buffer complete* callback for buffers provided to any device.

To start, pause, or stop tracing for a specific target device associated with the handle `device`, a tool calls the functions `ompt_target_start_trace`, `ompt_target_pause_trace`, or `ompt_target_stop_trace` with the following type signatures:

```
OMPT_TARG_API int ompt_target_start_trace(
  ompt_target_device_t *device,      /* target device handle      */
  ompt_target_buffer_request_callback_t request,
  ompt_target_buffer_complete_callback_t complete,
);

OMPT_TARG_API int ompt_target_pause_trace(
  ompt_target_device_t *device,      /* target device handle      */
  ompt_bool begin_pause              /* true=begin, false=end     */
);

OMPT_TARG_API int ompt_target_stop_trace(
  ompt_target_device_t *device       /* target device handle      */
);
```

A call to `ompt_target_stop_trace` also implicitly requests that the device flush any buffers that it has in its possession.

6.4 Processing Trace Records in a Buffer

There are several routines that need to be used together to process event records deposited in a buffer by a device. Function `ompt_target_advance_buffer_cursor`, with the type signature shown below, advances from one record in the specified buffer to the next.

```
OMPT_TARG_API int ompt_target_advance_buffer_cursor(
  ompt_target_buffer_t *buffer,
  ompt_target_buffer_cursor_t current,
  ompt_target_buffer_cursor_t *next
);
```

It returns 0 if the advance is successful and 1 if `current` indicates a position before the first record of the buffer or a position at or beyond the last record in the buffer.

The function `ompt_target_buffer_get_record_type` enables a tool to inspect the type of a record to determine whether it is both a known type and worth processing further.

```
OMPT_TARG_API ompt_record_type_t ompt_target_buffer_get_record_type(
  ompt_target_buffer_t *buffer,
  ompt_target_buffer_cursor_t current
);
```

`ompt_target_buffer_get_record_type` returns either `ompt_record_ompt` if the record represents a OMPT event, `ompt_record_correlation` if the purpose of the record is to associate the ID of an activity on a device with a host target task id, `ompt_record_native` if the record represents a device native record type that does not represent an OMPT event record, or `ompt_record_invalid` if the cursor is out of bounds.

Appendix A.6 defines the corresponding enumeration type for `ompt_record_type_t`. Section 6.5 describes the interface to use for accessing native record types.

To inspect the information for an OMPT record, a tool invokes the following function:

```
OMPT_TARG_API ompt_record_ompt_t *ompt_target_buffer_get_record_ompt(
  ompt_target_buffer_t *buffer,
  ompt_target_buffer_cursor_t current
);
```

This function returns a pointer that may point into the record buffer, or it may point into thread local storage where the information extracted from a record was assembled. The information available for an event depends upon its type. For this purpose, Appendix A.6 defines a union type that will be used to return information for different OMPT event record types. A subsequent call to `ompt_record_get` may overwrite the contents of the fields in the record returned by a prior invocation.

When a target task begins, it is assigned an ID. Each activity, e.g., a kernel invocation, that occurs on a device is assigned a device activity ID. All device activities can be mapped back to the target task that initiated them. Multiple device activities may be initiated by one target task; hence the mapping between device activities and target tasks is many to one. The mechanism for relating a device activity back to its associated target task is a correlation record. Correlation records appear among the stream of trace records from a target device. If a trace record is found to be of type `ompt_record_correlation`, then it may be extracted from the trace using the following function:

```
OMPT_TARG_API ompt_record_correlation_t *ompt_target_buffer_get_record_correlation(
  ompt_target_buffer_t *buffer,
  ompt_target_buffer_cursor_t current
);
```

6.5 Processing Native Trace Records in a Buffer

To inspect a native trace record for a device, a tool invokes the following function to obtain a pointer to a trace record in the native format for the device associated with buffer:

```
OMPT_TARG_API void *ompt_target_buffer_get_record_native(
  ompt_target_buffer_t *buffer,
  ompt_target_buffer_cursor_t current
);
```

The pointer returned may point into the trace buffer, or it may point into thread local storage where the information extracted from a trace record was assembled. The information available for a native event depends upon its type. A subsequent call to `ompt_target_buffer_get_record_native` may overwrite the contents of the fields in the record returned by a prior invocation.

Function `ompt_target_buffer_get_record_native_abstract` can be used by a tool to acquire basic information about a native record and handle it even without full support for native records from the associated device.

```
OMPT_TARG_API ompt_record_native_abstract_t *
ompt_target_buffer_get_record_native_abstract(
  void *native_record
);
```

A `ompt_target_buffer_get_record_native_abstract_t` record contains several pieces of information that a tool can use to process a native record that it may not fully understand. The record `rclass` field indicates whether the record is informational (`ompt_record_native_class_info`) or represents an event (`ompt_record_native_class_event`). Knowing whether a record is informational or represents an event can help a tool determine how to present the record. The record `type` field is points to a statically-allocated, immutable character string that provides a meaningful name a tool might want to use to describe the event to a user. The `start_time` and `end_time` fields are used to place an event in time. The times are relative to the device clock. If an event has no associated `start_time` and/or `end_time`, the value of an unavailable field will be the distinguished value `ompt_time_none`. The hardware id field, `hwid`, is used to indicate the location on the device where the event occurred. A `hwid` may represent a hardware abstraction such as a core or a hardware thread id. The precise semantics of a `hwid` value for a device is defined by the implementer of the software stack for the device. If there is no `hwid` associated with a record, the value of `hwid` shall be `ompt_hwid_none`. Finally, `target_task_id` indicates the target task with which this record is associated. If there is no associated target task or the associated target task is unknown, `target_task_id` will be 0.

omp-tool-var value	action
enabled	the OpenMP runtime will call `ompt_tool` before initializing itself.
disabled	the OpenMP runtime will not call `ompt_tool`, regardless of whether a tool is present or not.

Table 5: OpenMP runtime responses to settings of the *omp-tool-var* ICV.

7 Initializing OMPT Support for Tools

Section 7.1 describes how an OpenMP runtime uses the value of the *omp-tool-var* ICV to decide whether or not an OpenMP runtime will try to register a performance tool prior to runtime initialization. Section 7.2 explains how a registered performance tool initializes itself.

7.1 Tool Registration

The `OMP_TOOL` environment variable sets the *omp-tool-var* ICV, which controls whether or not an OpenMP runtime will try to register a performance tool or not. The value assigned to `OMP_TOOL` is case insensitive and may have leading and trailing white space. The value of this environment variable must be `enabled` or `disabled`. If `OMP_TOOL` is set to any value other than `enabled` or `disabled`, an OpenMP runtime will print a fatal error to the standard error file descriptor indicating that an illegal value had been supplied for `OMP_TOOL` and the program's execution will terminate. If `OMP_TOOL` is not defined, the default value for *omp-tool-var* is `enabled`.

Table 5 describes the action that an OpenMP runtime will take in response to possible values of *omp-tool-var*. If the value of *omp-tool-var* is `enabled`, the runtime will attempt to register a performance tool by calling the function `ompt_tool` before performing runtime initialization. The signature for `ompt_tool` is shown below:

```
extern "C" {
  ompt_initialize_fn_t ompt_tool(void);
};
```

If an OpenMP runtime calls `ompt_tool` but no tool-provided implementation of `ompt_tool` is present, a weak version of `ompt_tool` provided by the runtime will return NULL. If a tool-provided version of `ompt_tool` is present, it may return NULL indicating that the tool declines to register itself with the runtime; otherwise, the tool may register itself with the runtime by returning a non-NULL pointer to a function with type signature `ompt_initialize_fn_t`. The type signature for `ompt_initialize_fn_t` is described in Section 7.2. Since only one tool-provided definition of `ompt_tool` will be seen by an OpenMP runtime, only one tool may register itself. If a tool-supplied implementation of `ompt_tool` returns a non-NULL initializer, the OpenMP runtime will maintain state information for each OpenMP thread and will perform OMPT event callbacks registered during tool initialization.

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

7.2 Tool Initialization

When an OpenMP runtime receives a non-NULL pointer to a tool initializer function with signature `ompt_initialize_fn_t` as the result of a call to `ompt_tool`, the runtime will call the tool initializer immediately after the runtime fully initializes itself. The initializer must be called before beginning execution of any OpenMP construct or completing any execution environment routine invocation. The signature for the tool initializer callback is shown below:

```
typedef void (*ompt_initialize_fn_t) (
  ompt_function_lookup_t lookup,
  const char *runtime_version,
```

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 6: Meaning of return codes for `ompt_set_callback`.

```
    unsigned int ompt_version
  );
```

The second argument to `ompt_initialize` is a version string that unambiguously identifies an OpenMP runtime 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 implementation. Knowing exactly what version of an OpenMP runtime 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 2.

The two principal duties of a tool initializer are looking up pointers to all OMPT API functions that the tool uses and to registering tool callbacks. These two operations are described below.

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

```
  typedef ompt_interface_fn_t (*ompt_function_lookup_t)(
    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 not be visible to a preloaded tool. Within a tool, one uses `lookup` to obtain function pointers to each function in the OMPT API. All functions in the OMPT API are marked with `OMPT_API`. These functions should not be global symbols in an OpenMP runtime implementation to avoid tempting tool developers to call them directly.

Below, we show how to use the `lookup` function to obtain a pointer to the OMPT API function `ompt_get_thread_id`:

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

Other functions in the OMPT API may be looked up analogously. If a named function is not available in an OpenMP runtime's implementation of OMPT, `lookup` will return NULL.

Registering Callbacks. Tools register callbacks to receive notification of various events that occur as an OpenMP program executes using the OMPT API function `ompt_set_callback`. The signature for this function is shown below

```
  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`. The possible return codes for `ompt_set_callback` and their meaning is shown in Table 6. 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 OMPT API function `ompt_get_callback` may be called at any time to inspect whether a callback has been registered or not.

```
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.

8 Tool Control for Applications

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:

```
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.

9 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.

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.

References

[1] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. Sun Microsystems, Inc.. OpenMP ARB White Paper. Available online at http://www.compunity.org/futures/omp-api.html.

[2] G. Jost, O. Mazurov, and D. an Mey. Adding new dimensions to performance analysis through user-defined objects. In *Proceedings of the 2005 and 2006 International Conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 255–266, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.

[4] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 229–240, New York, NY, USA, 2009. ACM.

[5] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

A OMPT Interface Type Definitions

A.1 Runtime States

When OMPT is enabled, an OpenMP runtime 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 2.

```
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_critical          = 0x62, /* waiting for critical       */
  ompt_state_wait_atomic            = 0x63, /* waiting for atomic         */
  ompt_state_wait_ordered           = 0x64, /* waiting for ordered        */

  /* target (112..127) */
  ompt_state_wait_target            = 0x70, /* waiting for target         */
  ompt_state_wait_target_data       = 0x71, /* waiting for target data    */
  ompt_state_wait_target_update     = 0x72, /* waiting for target update  */

  /* misc (128..143) */
  ompt_state_undefined              = 0x80, /* undefined thread state     */
  ompt_state_first                  = 0x81, /* 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. The purpose of these callbacks is described in Section 3.

```
typedef enum {
  /*--- Mandatory Events ---*/
  ompt_event_thread_begin            = 5,  /* thread begin              */
  ompt_event_thread_end              = 6,  /* thread end                */

  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_target_task_begin       = 62, /* target task launch        */
  ompt_event_target_task_end         = 63, /* target task end (sync)    */

  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_worksharing_begin         = 35, /* task at worksharing begin    */
  ompt_event_worksharing_end            = 36, /* task at worksharing 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_flush                    = 61, /* after executing flush      */

  ompt_event_target_data_begin        = 65, /* target data begin          */
  ompt_event_target_data_end          = 66, /* target data end            */

  ompt_event_target_data_map_begin    = 67, /* before data mapping        */
  ompt_event_target_data_map_end      = 68, /* after data mapping         */

  ompt_event_task_dependences         = 69, /* new task dependences        */
  ompt_event_task_dependence_pair     = 70  /* new task dependence pair    */
} ompt_event_t;
```

A.3 Miscellaneous Type Definitions

This section describes miscellaneous enumeration types used by tool callbacks.

```
#define OMPT_API                            /* used to mark OMPT functions obtained from     *
                                             * lookup function passed to ompt_initialize     */

#define OMPT_TARG_API                       /* used to mark OMPT functions obtained from     *
                                             * lookup function passed to                     *
                                             * ompt_target_get_device_info                   */

typedef uint64_t ompt_thread_id_t;          /* uniquely identifies the thread                */

typedef uint64_t ompt_task_id_t;            /* uniquely identifies the task instance         */

typedef uint64_t ompt_parallel_id_t;        /* uniquely identifies the parallel instance     */

typedef uint64_t ompt_wait_id_t;            /* identify what a thread is awaiting            */

typedef uint64_t ompt_target_activity_id_t; /* ID of an activity on a device                 */

typedef uint32_t ompt_lock_impl_t;          /* small integer encoding the impl of a lock     */

typedef uint32_t ompt_lock_hint_t;          /* as described in the OpenMP language standard  */

typedef uint32_t ompt_bool;                 /* takes the values 0 (false) or 1 (true)        */

typedef void ompt_target_device_t;          /* opaque object representing a target device    */

typedef uint64_t ompt_target_time_t;        /* raw time value on a device                    */

typedef void ompt_target_buffer_t;          /* opaque handle for a target buffer             */

typedef uint64_t ompt_target_buffer_cursor_t; /* opaque handle for position in target buffer */

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

typedef enum ompt_scope_endpoint_e {
  ompt_scope_begin     = 1,
  ompt_scope_end       = 2
} ompt_scope_endpoint_t;

typedef enum ompt_sync_region_kind_e {
  ompt_sync_region_barrier,
  ompt_sync_region_taskwait,
  ompt_sync_region_taskgroup
} ompt_sync_region_kind_t;

typedef enum ompt_worksharing_type_e {
  ompt_worksharing_loop,
  ompt_worksharing_sections,
  ompt_worksharing_single,
  ompt_worksharing_workshare,
} ompt_worksharing_type_t;

typedef enum ompt_mutex_kind_e {
  ompt_mutex            = 0x10,
  ompt_mutex_lock       = 0x11,
  ompt_mutex_nest_lock  = 0x12,
  ompt_mutex_critical   = 0x13,
  ompt_mutex_atomic     = 0x14,
  ompt_mutex_ordered    = 0x20
} ompt_mutex_kind_t;

typedef enum ompt_target_task_type_e {
  ompt_target_task_target     = 1,
  ompt_target_task_enter_data = 2,
  ompt_target_task_exit_data  = 3,
  ompt_target_task_update     = 4
} ompt_target_task_type_t;

typedef enum ompt_native_mon_flags_e {
  ompt_native_data_motion_explicit     = 1,
  ompt_native_data_motion_implicit     = 2,
  ompt_native_kernel_invocation        = 4,
  ompt_native_kernel_execution         = 8,
  ompt_native_driver                   = 16,
  ompt_native_runtime                  = 32,
  ompt_native_overhead                 = 64,
  ompt_native_idleness                 = 128
} ompt_native_mon_flags_t;

typedef enum  ompt_task_type_e {
  ompt_task_initial     = 1,
  ompt_task_implicit   = 2,
  ompt_task_explicit   = 3,
  ompt_task_target     = 4,
  ompt_task_degenerate = 5
} ompt_task_type_t;

typedef enum ompt_invoker_e {
  ompt_invoker_program = 0,         /* program invokes master task  */
  ompt_invoker_runtime = 1          /* runtime invokes master task  */
} ompt_invoker_t;

typedef enum ompt_task_dependence_flag_e {
  // a two bit field for the dependence type
  ompt_task_dependence_type_out   = 1,
  ompt_task_dependence_type_in    = 2,
  ompt_task_dependence_type_inout = 3,
} ompt_task_dependence_flag_t;

typedef struct ompt_task_dependence_s {
  void *variable_addr;
  uint32_t  dependence_flags;
```

```
} ompt_task_dependence_t;

typedef enum ompt_target_map_flag_e {
  // a three bit field 0..7 for map type to/from/tofrom/alloc/release/delete
  ompt_target_map_flag_to      = 1,
  ompt_target_map_flag_from    = 2,
  ompt_target_map_flag_tofrom  = 3,
  ompt_target_map_flag_alloc   = 4,
  ompt_target_map_flag_release = 5,
  ompt_target_map_flag_delete  = 6,
  // 7 unused

  // one bit for synchronous/asynchronous
  ompt_target_map_flag_sync = 8,
} ompt_target_map_flag_t;

typedef struct ompt_frame_s {
  void *exit_runtime_frame;    /* next frame is user code     */
  void *reenter_runtime_frame; /* user frame that reenters the runtime  */
} ompt_frame_t;

#define ompt_hwid_none (-1)
#define ompt_dev_task_none (~0ULL)
#define ompt_time_none (~0ULL)

#define ompt_lock_type_first 0
#define ompt_atomic_type_first 0
```

A.4 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 3 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 void (*ompt_thread_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_scoped_sync_region_callback_t) (
  ompt_sync_region_kind_t kind,
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_parallel_id_t parallel_id,   /* ID of parallel region        */
  ompt_task_id_t  task_id,          /* ID of task                   */
  const void *codeptr_ra            /* return address of api call   */
);


typedef void (*ompt_mutex_release_callback_t) (
  ompt_mutex_kind_t kind,          /* kind of release, e.g. lock  */
  ompt_wait_id_t wait_id           /* wait ID                     */
);

typedef void (*ompt_scoped_mutex_callback_t) (
  ompt_mutex_kind_t kind,          /* kind of release, e.g. lock  */
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_wait_id_t wait_id           /* wait ID                     */
);

typedef void (*ompt_scoped_nested_lock_callback_t) (
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_wait_id_t wait_id           /* wait ID                     */
);



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

```
typedef void (*ompt_atomic_callback_t) ( /* for wait                */
  ompt_wait_id_t wait_id,           /* wait ID                     */
  uint32_t atomic_type              /* type of atomic operation    */
);

typedef void (*ompt_sync_callback_t) (
  ompt_parallel_id_t parallel_id,   /* ID of parallel region       */
  ompt_task_id_t  task_id,          /* ID of task                  */
  const void *codeptr_ra            /* return address of api call  */
);

typedef void (*ompt_worksharing_begin_callback_t) (
  ompt_worksharing_type_t wstype,
  ompt_parallel_id_t parallel_id,   /* ID of parallel region       */
  ompt_task_id_t  task_id,          /* ID of task                  */
  const void *codeptr_ra            /* return address of api call  */
);

typedef void (*ompt_worksharing_end_callback_t) (
  ompt_worksharing_type_t wstype,
  ompt_parallel_id_t parallel_id,   /* ID of parallel region       */
  ompt_task_id_t  task_id,          /* ID of task                  */
);

typedef void (*ompt_parallel_begin_callback_t) ( /* for new parallel  */
  ompt_task_id_t parent_task_id,    /* ID of parent task           */
  const ompt_frame_t *parent_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 */
  const void *codeptr_ofn,          /* pointer to outlined function */
  ompt_invoker_t invoker            /* who invokes master task?    */
);

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_parallel_end_callback_t) (
  ompt_parallel_id_t parallel_id,   /* ID of parallel region       */
  ompt_task_id_t task_id,           /* ID of task                  */
  ompt_invoker_t invoker            /* who invokes master task?    */
);

typedef void (*ompt_scoped_parallel_callback_t) (
  ompt_scope_endpoint_t endpoint,    /* begin or end                */
  ompt_parallel_id_t parallel_id,   /* ID of parallel region       */
  ompt_task_id_t  task_id          /* ID of task                  */
);

typedef void (*ompt_scoped_task_callback_t) (
  ompt_scope_endpoint_t endpoint    /* begin or end                */
  ompt_task_id_t  task_id           /* ID of task                  */
```

```
);

typedef void (*ompt_scoped_callback_t) (
  ompt_scope_endpoint_t endpoint    /* begin or end                 */
);

typedef void (*ompt_task_begin_callback_t) ( /* for new tasks       */
  ompt_task_id_t parent_task_id,    /* ID of parent task            */
  const ompt_frame_t *parent_frame, /* frame data for parent task   */
  ompt_task_id_t new_task_id,       /* ID of created task           */
  const void *codeptr_ofn           /* pointer to outlined function */
);

typedef void (*ompt_target_task_begin_callback_t) (
  ompt_task_id_t parent_task_id,    /* ID of parent task            */
  const ompt_frame_t *parent_frame, /* frame data for parent task   */
  ompt_task_id_t host_task_id,      /* ID of target task on host    */
  int device_id,                    /* ID of the device             */
  const void *target_task_code,     /* ptr to target code           */
  ompt_target_task_type_t task_type /* the type of the target task  */
);

typedef void (*ompt_target_task_end_callback_t) (
  ompt_task_id_t host_task_id       /* ID of target task on host    */
);

/* lock initialization */
typedef void (*ompt_lock_callback_t) (
  ompt_bool is_nested,   /* nested lock or not */
  ompt_wait_id_t wait_id,           /* wait ID                      */
  ompt_lock_hint_t hint,       /* OMP lock hint                */
  ompt_lock_impl_t impl        /* implementation          */
);

/* target device */
typedef void (*ompt_target_data_begin_callback_t) (
  ompt_task_id_t task_id,           /* ID of encountering task      */
  int device_id,                    /* ID of the device             */
  const void *codeptr_ra            /* return address of api call   */
);

typedef struct ompt_target_map_entry_s {
  void *host_addr;                  /* host  address of the data    */
  void *device_addr;                /* device address of the data   */
  size_t bytes;                     /* number of bytes mapped       */
  uint32_t mapping_flags;           /* sync/async, to/from          */
} ompt_target_map_entry_t;

typedef void (*ompt_target_data_map_begin_callback_t) (
  ompt_task_id_t task_id,           /* ID of encountering task      */
  int device_id,                    /* ID of the device             */
  const ompt_target_map_entry_t *items, /* items to be mapped       */
  uint32_t nitems,                  /* # of items to be mapped      */
  ompt_target_activity_id_t  map_id /* ID for map event             */
```

```
);

typedef void (*ompt_target_data_map_end_callback_t) (
  int device_id,                   /* ID of the device             */
  ompt_target_activity_id_t  map_id /* ID for map event             */
);

/* task dependences */
typedef void (*ompt_task_dependences_callback_t) (
  ompt_task_id_t task_id,           /* ID of task with dependences */
  const ompt_task_dependence_t *deps,/* vector of task dependences  */
  int ndeps                         /* number of dependences       */
);

typedef void (*ompt_task_pair_callback_t) (
  ompt_task_id_t first_task_id,
  ompt_task_id_t second_task_id
);

/* 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
);

/* target trace buffer management routines */
typedef void (*ompt_target_buffer_request_callback_t) (
  ompt_target_buffer_t** buffer,    /* pointer to host memory to store target records */
  size_t *bytes                    /* buffer size in bytes */
);

typedef void (*ompt_target_buffer_complete_callback_t) (
  int device_id,                    /* target device                                  */
  const ompt_target_buffer_t *buffer,/* pointer to buffer with target event records   */
  size_t bytes,                     /* number of valid bytes in the buffer            */
  ompt_target_buffer_cursor_t begin, /* position of first record                      */
  ompt_target_buffer_cursor_t end    /* position after last record                    */
);
```

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.5 OMPT Inquiry and Control API

The functions in this section are not global function symbols in an OpenMP runtime. These functions can be looked up by name using the `ompt_function_lookup_t` function passed to `ompt_initialize`, as described in Section 7.2 and Appendix A.7.

```
/* 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                      */
);

/* lock type inquiry */
OMPT_API int ompt_enumerate_lock_types( /* extract the set of lock types supported        */
  ompt_lock_type_t current_type,/* current lock type in the enumeration                  */
  ompt_lock_type_t *next_type,  /* next lock type in the enumeration                     */
  const char **next_type_name   /* string description of next lock type                  */
);

/* 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               */
);

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

/* task inquiry */
OMPT_API ompt_bool ompt_get_task_info(
  int ancestor_level,            /* how many levels removed from the current task        */
  ompt_task_type_t *type,        /* return the type of the task                          */
  ompt_task_id_t *task_id,       /* return the ID of the task                            */
  ompt_frame_t **task_frame,     /* return the task_frame of the task                      */
  ompt_parallel_id_t *par_id     /* return the ID of the parallel region                   */
);

/* target device inquiry */
OMPT_API int ompt_target_get_device_id(         /* return active target device ID        */
  void
);

OMPT_TARG_API ompt_target_time_t ompt_target_get_time( /* return current time on device  */
  ompt_target_device_t *device   /* target device handle                                   */
);

OMPT_TARG_API double ompt_target_translate_time(
  ompt_target_device_t *device,  /* target device handle                                   */
  ompt_target_time_t time
);

/* target tracing control */
OMPT_TARG_API int ompt_target_set_trace_ompt(
  ompt_target_device_t *device,  /* target device handle                                   */
  ompt_bool enable,              /* enable or disable                                      */
  ompt_record_type_t rtype       /* a record type                                          */
);

OMPT_TARG_API int ompt_target_set_trace_native(
  ompt_target_device_t *device,  /* target device handle                                   */
  ompt_bool enable,              /* enable or disable                                      */
  uint32_t  flags                /* event classes to monitor                               */
);

OMPT_TARG_API int ompt_target_start_trace (
  ompt_target_device_t *device,  /* target device handle                                   */
  ompt_target_buffer_request_callback_t request,  /* fn pointer to request trace buffer  */
  ompt_target_buffer_complete_callback_t complete /* fn pointer to return trace buffer   */
);

OMPT_TARG_API int ompt_target_pause_trace(
  ompt_target_device_t *device,  /* target device handle                                   */
  ompt_bool begin_pause
);

OMPT_TARG_API int ompt_target_stop_trace(
  ompt_target_device_t *device   /* target device handle                                   */
);

/* target trace record processing */
OMPT_TARG_API int ompt_target_advance_buffer_cursor(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                   */
  ompt_target_buffer_cursor_t current, /* cursor identifying position in buffer            */
  ompt_target_buffer_cursor_t *next    /* pointer to new cursor for next position          */
);

OMPT_TARG_API ompt_record_type_t ompt_target_buffer_get_record_type(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                 */
  ompt_target_buffer_cursor_t current  /* cursor identifying position in buffer          */
);

OMPT_TARG_API ompt_record_ompt_t *ompt_target_buffer_get_record_ompt(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                 */
  ompt_target_buffer_cursor_t current  /* cursor identifying position in buffer          */
);

OMPT_TARG_API ompt_record_correlation_t *ompt_target_buffer_get_record_correlation(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                 */
  ompt_target_buffer_cursor_t current  /* cursor identifying position in buffer          */
);

OMPT_TARG_API void *ompt_target_buffer_get_record_native(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                 */
  ompt_target_buffer_cursor_t current  /* cursor identifying position in buffer          */
);

OMPT_TARG_API ompt_record_native_abstract_t *
ompt_target_buffer_get_record_native_abstract(
  void *native_record                  /* pointer to native trace record                 */
);
```

A.6 Record Types and Buffer Management

```
/* OMPT record type */
typedef enum {
  ompt_record_ompt,
  ompt_record_correlation,
  ompt_record_native,
  ompt_record_invalid
} ompt_record_type_t;

typedef enum {
  ompt_record_native_class_info = 0,
  ompt_record_native_class_event = 1
} ompt_record_native_class_t;

/* native record abstract */
typedef struct ompt_record_native_abstract_s {
  ompt_record_native_class_t rclass;
  const char *type;
  ompt_target_time_t start_time;
  ompt_target_time_t end_time;
  ompt_target_activity_id_t dev_task_id;
  uint64_t hwid;
} ompt_record_native_abstract_t;

/* correlation */
typedef struct ompt_record_correlation_s {
  ompt_task_id_t host_task_id;
  ompt_target_activity_id_t dev_task_id;
} ompt_record_correlation_t;

/* record types */
typedef struct ompt_record_thread_s {
  ompt_thread_id_t thread_id;      /* ID of thread                  */
} ompt_record_thread_t;

typedef struct ompt_record_thread_type_s {
  ompt_thread_type_t thread_type;  /* type of thread                */
  ompt_thread_id_t thread_id;      /* ID of thread                  */
} ompt_record_thread_type_t;

typedef struct ompt_record_wait_id_s {
  ompt_wait_id_t wait_id;          /* wait ID                       */
} ompt_record_wait_id_t;

typedef struct ompt_record_parallel_s {
  ompt_parallel_id_t parallel_id;  /* ID of parallel region         */
  ompt_task_id_t task_id;          /* ID of task                    */
} ompt_record_parallel_t;

typedef struct ompt_record_workshare_begin_s {
  ompt_parallel_id_t parallel_id;  /* ID of parallel region          */
  ompt_task_id_t task_id;          /* ID of task                     */
  void *codeptr_ra;                /* runtime call return address    */
```

```
} ompt_record_workshare_begin_t;

typedef struct ompt_record_parallel_begin_s {
  ompt_task_id_t parent_task_id;   /* ID of parent task              */
  ompt_frame_t *parent_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 *codeptr_ofn;               /* pointer to outlined function   */
} ompt_record_parallel_begin_t;

typedef struct ompt_record_task_s {
  ompt_task_id_t task_id; /* ID of task */
} ompt_record_task_t;

typedef struct ompt_record_task_pair_s {
  ompt_task_id_t first_task_id;
  ompt_task_id_t second_task_id;
} ompt_record_task_pair_t;

typedef struct ompt_record_task_begin_s {
  ompt_task_id_t parent_task_id;   /* ID of parent task              */
  ompt_frame_t *parent_task_frame; /* frame data of parent task      */
  ompt_task_id_t new_task_id;      /* ID of created task             */
  void *codeptr_ofn;               /* pointer to outlined function   */
} ompt_record_task_begin_t;

typedef struct ompt_record_target_task_begin_s {
  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 target_task_id;    /* ID of target task             */
  int device_id;                    /* ID of the device              */
  void *target_task_code;           /* pointer to target task code   */
} ompt_record_target_task_begin_t;

typedef struct ompt_record_target_data_begin_s {
  ompt_task_id_t task_id;           /* ID of encountering task       */
  int device_id;                    /* ID of the device              */
  void *codeptr_ra;                 /* return address of api call    */
} ompt_record_target_data_begin_t;

typedef struct ompt_record_data_map_begin_s {
  ompt_task_id_t task_id;           /* ID of encountering task       */
  int device_id;                    /* ID of the device              */
  void *host_addr;                  /* host  address of the data     */
  void *device_addr;                /* device address of the data    */
  size_t bytes;                     /* number of bytes mapped        */
  uint32_t mapping_flags;           /* sync/async, to/from           */
  void *target_map_code;            /* ptr to target map code        */
} ompt_record_data_map_begin_t;

typedef struct ompt_record_data_map_done_s {
  ompt_task_id_t task_id;           /* ID of current task            */
  int device_id;                    /* ID of the device              */
  void *host_addr;                  /* host  address of the data     */
  void *device_addr;                /* device address of the data   */
  size_t bytes;                     /* number of bytes mapped       */
  uint32_t mapping_flags;           /* sync/async, to/from          */
} ompt_record_data_map_done_t;

typedef struct ompt_record_task_dependences_s {
  ompt_task_id_t task_id;           /* ID of task with dependences  */
  ompt_task_dependence_t *deps;     /* vector of task dependences   */
  int ndeps;                        /* number of dependences        */
} ompt_record_task_dependences_t;

/* OMPT record */
typedef struct ompt_record_ompt_s {
  ompt_event_t type;                /* event type                       */
  ompt_target_time_t time;          /* time record created              */
  ompt_thread_id_t thread_id;       /* thread ID for this record        */
  ompt_target_activity_id_t dev_task_id;  /* link to host context       */
  union
  {
    ompt_record_thread_t thread;                 /* for thread          */
    ompt_record_thread_type_t type;              /* for thread type     */
    ompt_record_wait_id_t waitid;                /* for wait            */
    ompt_record_parallel_t parallel;             /* for inside parallel */
    ompt_record_workshare_begin_t new_workshare; /* for workshares      */
    ompt_record_parallel_begin_t new_parallel;   /* for new parallel    */
    ompt_record_task_t task;                     /* for tasks           */
    ompt_record_task_pair_t task_switch;         /* for task switch     */
    ompt_record_task_begin_t new_task;           /* for new tasks       */
  } record;
} ompt_record_ompt_t;
```

A.7 Registration and Initialization

Function ompt_tool is the only global symbol associated with OMPT. To use OMPT, a tool overlays an implementation of ompt_tool in place of the default one provided by an OpenMP runtime. Use of this interface is described in Section 7.1.

```
extern "C" {
  ompt_initialize_fn_t ompt_tool(void);
};
```

Function ompt_tool returns a pointer to a tool initializer, whose type signature is described below. Use of this interface is described in Section 7.2.

```
typedef void (*ompt_initialize_fn_t) (
  ompt_function_lookup_t lookup,
  const char *runtime_version,
  unsigned int ompt_version
);
```

B Task Frame Management and Inspection

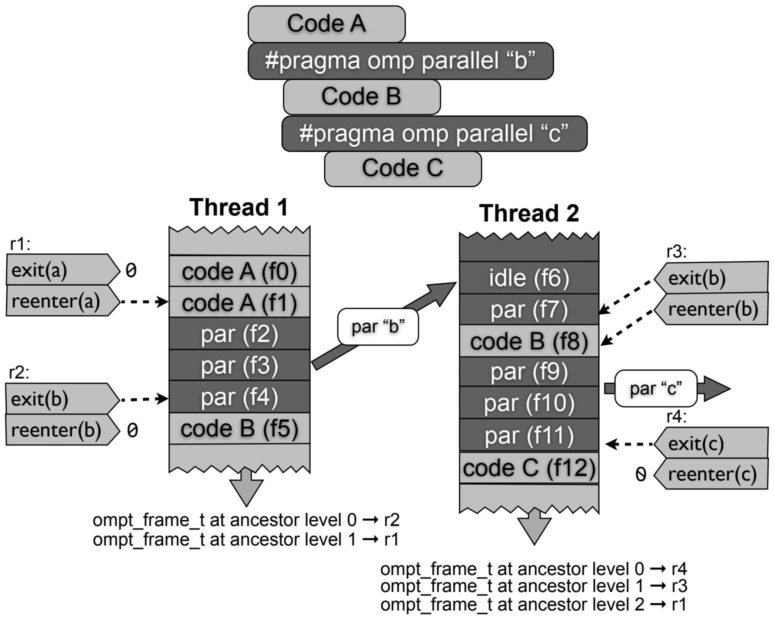

Figure 1: Frame information.

Figure 1 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 1 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 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 f1—the user frame in the initial task that calls the runtime. The `ompt_frame_t` for the initial task is labeled `r1` in Figure 1. 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.

Let us focus now on thread 2, an OpenMP thread. Figure 1 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. 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," as execution returns to the runtime, the runtime fills in the `reenter_runtime_frame` field of the current task's `ompt_frame_t` (labeled `r3`) to frame f8—the frame that invoked the runtime. Before the task for parallel region "c" is invoked on thread 2, the runtime system sets the `exit_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 1, the figure shows the `ompt_frame_t` information obtained by calls to `ompt_get_task_info` made on each thread for the stack state shown. We show the ID of the `ompt_frame_t` record returned at each ancestor level. Note that thread 2 has task frame information for three levels of tasks, whereas thread 1 has only two.

C Implementation Considerations for Tool Registration

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

- The tool implementation of `ompt_tool` is statically-linked into an executable. Such an implementation of `ompt_tool` will be visible to an OpenMP runtime regardless of whether the runtime is statically linked into the executable or dynamically-linked into a shared library.
- An implementation of `ompt_tool` is in a tool's shared library, which we denote $\mathcal{L}_T$. Such an implementation of `ompt_tool` will be visible to an OpenMP runtime 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 registration in the OpenMP runtime 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 to define `ompt_tool` as a weak global symbol that returns 0. Definition of `ompt_tool` 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_tool` 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_tool` 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_tool` in a separate object file. In this case, the linker will include the OpenMP library's stub implementation of `ompt_tool` 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_tool` as a global symbol. The version in the OpenMP library would be invoked only if no tool-supplied implementation of `ompt_tool` 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. 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 present in an executable or a shared library to invoke a tool-supplied version of a version of `ompt_tool` rather than the default implementation of `ompt_tool` present in the OpenMP runtime.

[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.