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 June 22, 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 allow 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 identifier per OpenMP thread, parallel region, task region, target region, and target operation. In addition, each thread maintains a wait identifier.
- 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 create, task schedule, implicit task begin/end, target region begin/end, target data operation, target submission, 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 6 describes an API for tracing activities on target devices. Section 7 describes the OMPT API operations for tool initialization. Section 8 describes the tool control interface available to applications. 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.

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 `omp_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 `omp_state_undefined`.

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 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 `omp_state_wait_lock` early before the thread attempts to acquire a lock. Another compliant runtime may transition a thread's state to `omp_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 `omp_state_wait_lock` even later, e.g., only after the thread waits for a significant amount of time.

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

Idle State

`omp_state_idle`

The thread is idle, waiting for work.

Work States

`omp_state_work_serial`

The thread is executing code outside all parallel regions.

`omp_state_work_parallel`

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

`omp_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 `omp_state_work_parallel` or `omp_state_overhead`.

Overhead State

`omp_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.

Barrier Wait States

`omp_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 `omp_state_wait_barrier_implicit` or `omp_state_wait_barrier_explicit`, as appropriate.

`omp_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 `omp_state_wait_barrier` for implicit barriers.

`omp_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 `omp_state_wait_barrier` for explicit barriers.

Task Wait States

`omp_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.

`omp_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

OpenMP provides several mechanisms that enforce mutual exclusion: locks, critical, atomic, and ordered. A runtime implementation may report a thread waiting for any type of mutual exclusion using either a state that precisely identifies the type of mutual exclusion, or a more generic state such as `omp_state_wait_mutex` or `omp_state_wait_lock`. This flexibility may significantly simplify the maintenance of states associated with mutual exclusion in the runtime when various mechanisms for mutual exclusion rely on a common implementation, e.g., locks.

`omp_state_wait_mutex`

The thread is waiting for a mutex of an unspecified type. A compliant implementation may have a thread enter this state early, when a thread encounters a lock acquisition or a region that requires mutual exclusion, or late, when the thread begins to wait.

`omp_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.

`omp_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.

`omp_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 opt not to report this state, for example, when using atomic hardware instructions that support non-blocking atomic implementations.

`omp_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.

Target Wait States

`omp_state_wait_target`

The thread is waiting for a target region to complete.

`omp_state_wait_target_data`

The thread is waiting for a target data mapping operation to complete. A compliant runtime implementation may report `omp_state_wait_target` for target data constructs.

`omp_state_wait_target_update`

The thread is waiting for a target update operation to complete. A compliant runtime implementation may report `omp_state_wait_target` for target update constructs.

Undefined

`omp_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.

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. 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_begin_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_end_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_create`

The OpenMP runtime invokes this callback upon encountering a task construct or a target construct that causes a task to be created. The callback executes in the context of the task that encountered the task or target construct. This callback has type signature `ompt_task_create_callback_t`. The callback argument `type` may indicate either an explicit task or one of the varieties of target tasks. The callback argument `has_dependences` is true if the task has dependences with respect to data objects.

`ompt_event_task_schedule`

The OpenMP runtime invokes this callback after it completes or suspends one task and before it schedules another task. This callback executes in the context of the newly-scheduled task. This callback has type signature `ompt_task_schedule_callback_t`. The callback argument `prior_task_data` indicates the prior task. The callback argument `prior_completed` is set if the prior task completed. The callback argument `next_task_data` indicates the task being scheduled.

`ompt_event_implicit_task`

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. The OpenMP runtime 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. This callback has type signature `ompt_scoped_implicit_callback_t`.

Target Regions

`ompt_event_target`

The OpenMP runtime invokes this callback with argument `endpoint=ompt_scope_begin` after a task encounters any target construct. The OpenMP runtime invokes this callback with argument `endpoint=ompt_scope_end` when the execution of this construct completes on the host. This callback executes in the context of the task that encounters the target construct. This callback has type signature `ompt_scoped_target_callback_t`. The callback argument `kind` indicates the kind of target construct. The callback argument `task_data` indicates the encountering task. The callback argument `device_id` indicates the device associated with the target construct. The callback argument `target_id` uniquely identifies a target construct instance. The `codeptr_ra` callback argument contains the return address of the call to the OpenMP runtime routine, which relates the target construct to the user program.

`ompt_event_target_data`

The OpenMP runtime invokes this callback prior to a transfer or delete operation and after an allocate operation. This callback occurs only if will result in activity on the target device. This callback has type signature `ompt_target_data_callback_t`. The callback argument `optype` indicates whether the data operation is allocate, transfer to device, transfer from device, or delete. The callback arguments `host_addr` and `device_addr` indicate the locations of the data on the host and device, respectively. The callback argument `size` indicates the number of data bytes. The callback argument `target_id` indicates the instance of the target construct associated with this operation. The callback argument `host_op_id` provides a unique host-side identifier that represents the activity on the device.

`ompt_event_target_submit`

The OpenMP runtime invokes this callback prior to submitting a kernel for execution on a target device. This callback has type signature `ompt_target_submit_callback_t`. The callback argument `target_id` indicates the instance of the target construct associated with this operation. The callback argument `host_op_id` provides a unique host-side identifier that represents the activity on the device. The callback arguments `requested_num_teams` `granted_num_teams` indicate, respectively, the number of teams requested by the user and granted by the runtime.

Application Tool Control

`ompt_event_control`

If the user program calls `ompt_control`, the OpenMP runtime invokes this callback. The callback executes in the context that the call occurs in the user program. This callback has type signature `ompt_control_callback_t`. Arguments passed to the callback are those passed by the user to `ompt_control`.

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 information gathered. 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. Section 3.2.1 describes a set of events intended primarily for use by sampling-based performance tools. These events enable a sampling-based performance tool to employ a strategy known as *blame shifting* to attribute waiting to activities that cause other threads to wait rather than to contexts in which waiting is observed. Section 3.2.2 describes additional events that, when used in conjunction with other events described in Section 3, enable a tool to receive notifications for all OpenMP runtime events. 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 designed for use by sampling-based performance tools that employ *blame shifting* to transfer blame for waiting from contexts where waiting is observed to activities responsible for the waiting.[2] The 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. The 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 the begin and end notifications for waiting.

`ompt_event_idle`

The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` when a thread waits for work outside a parallel region. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_end` before the thread begins to execute an implicit task for a parallel region or terminates. The callback executes in the environment of the waiting thread. This callback has type signature `ompt_idle_callback_t`.

[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].

`ompt_event_sync_region_wait`

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. The OpenMP runtime invokes this callback with the `endpoint=ompt_scope_end` when the task stops waiting in the region. This callback has type signature `ompt_scoped_sync_region_callback_t`. 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 callback argument `codeptr_ra` 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_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. The `wait_id` parameter identifies the lock or synchronization 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 so that 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 and require explicit retries include transactions, load-linked/store-conditional, or compare-and-swap.

3.2.2 Events for Instrumentation-based Measurement Tools (Optional)

The following events designed for instrumentation-based tools enable tools to receive notification for additional OpenMP runtime events of interest.

Tasking

`ompt_event_task_dependences`

If a task has any dependences with respect to data objects that constrain its ordering with respect to other tasks, the OpenMP runtime invokes this callback immediately after the callback announcing the task's creation 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 (`src_task_data`) and a consumer (`sink_task_data`) that blocked execution of the consumer. This callback will occur before the consumer knows that the dependence is satisfied. This may happen early or late. Note: this callback is used only to report blocking dependences between sibling tasks whose lifetimes overlap. No callback will occur if a producer task finishes before a consumer task is created. This callback has type signature `ompt_task_dependence_callback_t`.

Worksharing

`ompt_event_worksharing`

The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` after a task encounters a worksharing construct but before the task executes its first unit of work for the worksharing

region. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_end` 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 has type signature `ompt_scoped_worksharing_callback_t`. The `wstype` callback argument indicates whether the worksharing construct is a loop, sections, single executor or other participant, or workshare. The `codeptr_ra` callback argument contains the return address of the call to the OpenMP runtime routine, which relates the worksharing region to the user program, may be NULL when `endpoint=ompt_scope_end`. This callback executes in the context of the task that encountered the construct.

Master Blocks

`ompt_event_master`

The OpenMP runtime invokes this callback with `endpoint=ompt_scope_begin` after the implicit task of a master thread encounters a master construct but before the task executes the master region. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_end` after the implicit task of a master thread executed a master region but before the task executes the statement following the master construct. This callback has type signature `ompt_scoped_master_callback_t`. This callback executes in the context of the implicit task of a team's master thread.

Target Data Mapping

`ompt_event_target_data_map`

The OpenMP runtime invokes this callback when a set of `nitems` variables is mapped to or unmapped from the device data environment by a target, target data, target enter data or target exit data construct. This callback has type signature `ompt_target_data_map_callback_t`. The callback argument `target_id` indicates the instance of the target construct associated with this operation. The callback arguments `host_addr`, `device_addr`, `bytes`, and `mapping_flags` are arrays that describe data items mapped or unmapped. Elements of the `mapping_flags` array are bit vectors whose bits correspond to items in the enum `ompt_target_map_flag_t`. The callback executes in the context of the encountering task.

Barrier, Taskwait, and Taskgroup

`ompt_event_sync_region`

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

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 a lock initialization routine. This callback has type signature `ompt_lock_init_callback_t`. The callback argument `is_nest_lock` indicates the type of lock being initialized. The callback argument `wait_id` identifies the lock. The `hint` parameter is the lock

hint value passed to a hinted lock initialization routine. The `kind` parameter is a small integer indicating the lock implementation chosen by the OpenMP runtime. The mapping between values of `kind` and the lock implementations they represent can be determined using `ompt_enumerate_mutex_kinds`.

`ompt_event_destroy_lock`

The OpenMP runtime invokes this callback just before a task destroys a lock or nest lock. This callback has type signature `ompt_lock_destroy_callback_t`. The callback argument `wait_id` identifies the lock. This callback executes in the context of the task that called a lock destruction routine.

Lock, Nest Lock, Critical Section, Atomic, and Ordered

`ompt_event_mutex_acquire`

The OpenMP runtime invokes this callback when a task invokes `omp_set_lock` to acquire a lock, invokes `omp_set_nest_lock` to acquire a nest lock not already owned, or encounters a critical, atomic, or ordered construct. This callback has type signature `ompt_mutex_acquire_callback_t`. The callback argument `kind` indicates the kind of mutex being acquired. The callback argument `hint` is the implementation hint value specified for a (nest) lock, critical, or atomic construct. If no hint is available, e.g., for ordered constructs, `hint=omp_hint_unknown`. The callback argument `impl` indicates the implementation choice associated with a lock, nest lock, critical, or atomic, or ordered. The callback argument `wait_id` identifies the (nest) lock, a critical construct's associated *name* or synchronization variable, the program variable or synchronization variable associated with the atomic construct, or the synchronization variable associated with the ordered construct. This callback executes in the context of the task that called `omp_set_lock` or `omp_set_nest_lock` or encountered the critical, atomic, or ordered construct.

`ompt_event_mutex_acquired`

The OpenMP runtime invokes this callback just after the task acquires a (nest) lock or enters a critical, atomic, or ordered region. This callback has type signature `ompt_mutex_callback_t`. The callback argument `kind` indicates the kind of mutex being acquired. The callback argument `wait_id` identifies the (nest) lock, a critical construct's associated *name* or synchronization variable, or the program variable or synchronization variable associated with the atomic construct, or the synchronization variable associated with the ordered construct. This callback executes in the context of the task that called `omp_set_lock` or `omp_set_nest_lock` or encountered the critical or atomic construct.

`ompt_event_nested_lock`

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. The OpenMP runtime invokes this callback with `endpoint=ompt_scope_end` just after a task completes the nested acquisition. This callback has type signature `ompt_scoped_nested_lock_callback_t`. This callback executes in the context of the task that called an OMP API routine to set or unset a nest lock; its callback argument `wait_id` identifies the nest lock.

Miscellaneous

`ompt_event_flush`

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

4 Tool Data Structures

Threads, parallel regions, task regions, target regions, and target operations are represented by unique identifiers of type `ompt_data_t`. This type allows tools to either attach tool-specific data to the aforementioned

constructs or to maintain a tool-specifc integer ID. Both options require the runtime to pass this identifiers by reference to the corresponding callbacks. The initial value of an identifier is `ompt_data_none`. This allows tools to detect if the identifier has already been initialzed. Tools may assign the identifier a different value. It is the tool's responsibility to maintain the resources it assigns to the identifiers. If the runtime needs to report an invalid identifier, it passes a NULL pointer to the callback or returns a NULL pointer from an inquiry API function.

4.1 Thread Identifier

Each OpenMP thread has an associated identifier of type `ompt_data_t`. On thread creation, the runtime library initializes the thread identifier to `ompt_data_none`. Thread related event callbacks provide the identifier by reference in order to let a tool change its value. A thread identifier can be retrieved on demand by invoking the `ompt_get_thread_data`function (described in Section 5.3). To indicate an invalid identifier, this function returns a NULL pointer.

4.2 Parallel Region Identifiers

Each OpenMP parallel region has an associated identifier of type `ompt_data_t`. At the begin of a parallel region, the runtime library initializes the parallel region identifier to `ompt_data_none`. Parallel region related event callbacks provide the identifier by reference in order to let a tool change its value. A parallel region identifier can be retrieved on demand by invoking the `ompt_get_parallel_info` function (described in Section 5.4). To indicate an invalid identifier, this function returns a NULL pointer.

4.3 Task Region Identifiers

Each OpenMP task has an associated identifier of type `ompt_data_t`. Task identifiers are assigned to initial, implicit, explicit, and target tasks. On task region creation, the runtime library initializes the task region identifier to `ompt_data_none`. Task region related event callbacks provide the identifier by reference in order to let a tool change its value. A task region identifier can be retrieved on demand by invoking the `ompt_get_task_info` function (described in Section 5.5). To indicate an invalid identifier, this function returns a NULL pointer.

4.4 Target Region and Operation Identifiers

Each OpenMP target region and target operation has an associated identifier of type `ompt_id_t`. A unique target identifier is assigned on the host each time an instance of a target construct is encountered. Each operation within a target region, e.g., transferring data to/from a device or launching a kernel launch on a device, is also assigned a unique target identifier. Identifiers assigned to target regions or operations are unique from the time an OpenMP runtime is initialized until it is shut down. The current target region and operation identifiers can be retrieved by invoking the `ompt_get_target_info` function (described in Section 5.6). Tools should not assume that `ompt_id_t` values are small or densely allocated. The value `ompt_id_none` is reserved to indicate an invalid target identifier. The value `ompt_id_none` will be returned for (a) the target region identifier if `ompt_get_target_info` is invoked outside a target region and (b) the target operation identifier if `ompt_get_target_info` is invoked while no target operation is in progress.

4.5 Wait Identifiers

Each thread instance maintains a *wait identifier* of type `ompt_wait_id_t`. When a task executing on a thread is waiting for something, the thread's wait identifier indicates what the thread is awaiting. A wait identifier may represent a critical section *name*, a lock, a program variable accessed in an atomic region, or a synchronization object internal to an OpenMP runtime implementation. A thread's wait identifier can be retrieved on demand by invoking the `ompt_get_state` function (described in Section 5.3). Tools should not assume that `ompt_wait_id_t` values are small or densely allocated. When a thread is not in a wait state, a thread's wait identifier has an undefined value.

exit / enter	enter = null	enter = 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 schedule 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 scheduled an implicit, explicit, or target task

Table 1: Meaning of various values for `exit_frame` and `enter_frame`.

4.6 Structure to Support Classification of Stack Frames

When executing an OpenMP program, at times procedure frames from the OpenMP runtime appear on the call stack between user code procedure frames. To enable a tool to classify procedure frames on the call stack as belonging to the user program or the OpenMP runtime, 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 parallel region or task does not create a new task. A degenerate task may arise when a parallel construct is encountered in a parallel region and nested parallelism is not enabled or when an orphaned directive that would create a task is encountered outside a parallel region. 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*.

Each initial, implicit, explicit, or target task maintains an `ompt_frame_t` data structure that contains a pair of pointers.

```
typedef struct ompt_frame_s {
    void *exit_frame;  /* runtime frame that calls user code */
    void *enter_frame; /* user frame that calls the runtime  */
} ompt_frame_t;
```

An `ompt_frame_t`'s lifetime begins when a task is created and ends when the task is destroyed. Tools should not assume that a frame structure remains at a constant location in memory throughout a task's lifetime. Frame data is passed to some callbacks; it can also be retrieved asynchronously by invoking the `ompt_get_task_info` function (described in Section 5.5) in a signal handler. Frame data contains two components:

`exit_frame` This value is set before the OpenMP runtime invokes a procedure containing user code. This field points to the frame pointer of the runtime procedure frame that invoked the user code. For compilers that generate code where the master thread for a parallel region invokes user code directly (e.g., older versions of GNU compilers), this may point to a frame of user code for the enclosing task. This value is NULL until just before the runtime invokes a procedure containing user code.

`enter_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 a task region. This value is set when a task enters the runtime and cleared before the runtime returns control to the task.

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_frame` and `enter_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 into, or returns from, 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 should 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 may be reported by the runtime that is being used.

```
OMPT_API _Bool ompt_enumerate_states(
  omp_state_t current_state,
  omp_state_t *next_state,
  const char **next_state_name
);
```

To begin enumerating the states that a runtime system supports, the value `omp_state_undefined` should be supplied for `current_state` in the call to `ompt_enumerate_states` that begins the enumeration. The argument `next_state` is a pointer to an `omp_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_states` should pass the code returned in `next_state` by the prior call. Whenever one or more states are left in the enumeration, `ompt_enumerate_states` will return `true`. When the last state in the enumeration is passed to `ompt_enumerate_states` as `current_state`, the function will return `false` indicating that the enumeration is complete. An example of how to enumerate the states supported by an OpenMP runtime is shown below:

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

5.2 Enumerate Mutex Implementations

The OpenMP standard recognizes that a runtime system may implement locks, nest locks, critical sections, and atomic regions in several different ways. For that reason, a user program can provide hints to help the runtime system to select appropriate implementations. When a lock or nest lock is initialized, the `ompt_event_init_lock` callback receives the argument `kind`—a small integer that indicates the lock implementation chosen by the OpenMP runtime. Similarly, the `ompt_event_mutex_acquire` callback receives the argument `kind` to indicate the implementation of a lock, critical section, atomic region, or ordered section. To enable a tool to provide insight into a runtime system's implementation choices, OMPT provides the following interface for enumerating the synchronization implementations it employs.

```
OMPT_API _Bool ompt_enumerate_mutex_kinds(
```

```
  uint32_t current_kind,
  uint32_t *next_kind,
  const char **next_kind_name
);
```

To begin enumerating the mutex implementations that a runtime employs, one should supply `ompt_mutex_kind_unknown` as the value for `current_kind` in the first call to `ompt_enumerate_mutex_kinds`. The argument `next_kind` is a pointer to an integer that will be set to the code for the next implementation in the enumeration. The argument `next_kind_name` is a pointer to a location that will be filled in with a pointer to the name of the mutex implementation associated with `next_kind`. Each subsequent invocation of `ompt_enumerate_mutex_kinds` should pass the code returned in `next_kind` by the prior call. Whenever one or more mutex implementations are left in the enumeration, `ompt_enumerate_mutex_kinds` will return `true`. When the last mutex implementation type in the enumeration is passed to `ompt_enumerate_mutex_kinds` as `current_kind`, the function will return `false` indicating that the enumeration is complete. An example of how to enumerate the types of mutex implementations supported by an OpenMP runtime is shown below:

```
uint32_t kind = ompt_mutex_kind_unknown;
const char *kind_name;
while (ompt_enumerate_mutex_kinds(kind, &kind, &kind_name)) {
  // tool notes that the runtime "kind" is associated with "kind_name"
}
```

5.3 Thread Inquiry

Function `ompt_get_thread_data` returns the address of the thread identifier for the current thread.

```
OMPT_API ompt_data_t *ompt_get_thread_data(void);
```

If the current thread is unknown to the OpenMP runtime, the function returns NULL. A tool can check if the thread identifier is in its initial state by comparing against `ompt_data_none`. A tool may modify the identifier. *This function is async signal safe.*

Function `ompt_get_state` returns the state of the current thread. If the returned state indicates that the thread is waiting for a lock, nest lock, critical section, atomic region, or ordered region, it will fill in `*wait_id` with the wait identifier associated with the state. *This function is async signal safe.*

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

5.4 Parallel Region Inquiry

Function `ompt_get_parallel_info` returns information about the parallel region, if any, at the specified ancestor level in the current execution context.

```
OMPT_API _Bool ompt_get_parallel_info(
  int ancestor_level,
  ompt_data_t **parallel_data,
  int *team_size
);
```

Ancestor level 0 refers to the current parallel region; information about enclosing parallel regions may be obtained using higher ancestor levels. Function `ompt_get_parallel_info` returns a Boolean, which indicates whether there is a parallel region at the specified ancestor level. The function results `**parallel_data` and `*team_size` indicate properties of the parallel region at the specified ancestor level. If there is no enclosing parallel region at the specified ancestry level, the address of the provided parallel region identifier `*parallel_data` equals NULL and the value of `*team_size` is undefined. A tool can check if the parallel region identifier is in its initial state by comparing against `ompt_data_none`. A tool may modify the identifier. *This function is async signal safe.*

5.5 Task Region Inquiry

Function `ompt_get_task_info` provides information about the task, if any, at the specified ancestor level in the current execution context.

```
OMPT_API _Bool ompt_get_task_info(
  int ancestor_level,
  ompt_task_type_t *type,
  ompt_data_t **task_data,
  ompt_frame_t **task_frame,
  ompt_data_t **parallel_data,
  uint32_t *thread_num
);
```

Ancestor level 0 refers to the current task; information about ancestor tasks in the current execution context may be queried at higher ancestor levels. Function `ompt_get_task_info` returns a Boolean, which indicates whether there is a task at the specified ancestor level. *This function is async signal safe.*

A task may be of type initial, implicit, explicit, target, or degenerate. If the task at the specified level is degenerate, the address returned in `*task_data` will be NULL. A degenerate task will be associated with the enclosing parallel region. If the thread invoking this function is outside any parallel region, the address returned in `*parallel_data` will be NULL. A tool can check if the task region or parallel region identifier are in their initial states by comparing against `ompt_data_none`. A tool may modify the identifiers. The value returned in `thread_num` indicates the number of the OpenMP thread executing the task in the parallel region to which the task belongs.

Using values inside `ompt_frame_t` objects returned by calls to `ompt_get_task_info`, a tool 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 of the call stack back to a source-level view that is easier for application developers to understand. Appendix B discusses an example that illustrates the use of `ompt_frame_t` objects with multiple threads and nested parallelism.

5.6 Target Region Inquiry

Function `ompt_get_target_info` provides the identifier for a thread's current target region and target operation id, if any.

```
OMPT_API _Bool ompt_get_target_info(
  ompt_id_t *target_id,                    /* target region id          */
  ompt_id_t *host_op_id                    /* host side ID for operation */
);
```

This function returns `false` if the thread invoking it is outside a target region. In that case, the values of both of its returned parameters are undefined. If `ompt_get_target_info` returns `true`, the value for `*host_op_id` will be `ompt_id_none` if the thread invoking it is not in the process of initiating an operation on a target (e.g., copying data to or from an accelerator or launching a kernel). *This function is async signal safe.*

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.*

[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.

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 _Bool ompt_target_get_device_info(
  int32_t device_id,
  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 `true` indicating success; otherwise, it will return `false` and the values of its return parameters are undefined. 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 the specific device. The OpenMP 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 to calls to device-specific functions in the target interface to identify the device being addressed.

The OpenMP 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`. 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 host and target devices 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 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 time value obtained from a target device to a corresponding time value on the host. 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                         /* time value from device     */
);
```

Advice to tool implementers: The accuracy of time translations may degrade if they are not performed promptly after a device time value is received if either the host or device vary their clock speeds. Prompt translation of device times to host times is recommended.

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 2: Meaning of return codes for `ompt_trace_set_ompt` and `ompt_target_set_trace_native`.

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 the design of a performance monitoring 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 the 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      */
  _Bool enable,                                    /* enable or disable         */
  ompt_event_t etype                               /* an event type             */
);
```

The argument `device` is a handle to identify the target device whose performance monitoring may be altered by invoking this function. The boolean `enable` indicates whether recording of events of type `rtype` should 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 2 shows the possible return codes for `ompt_target_set_trace_ompt`. If a single invocation of `ompt_target_set_trace_ompt` is used to enable or disable more than one event (i.e., `rtype`=0), the return code will be 3 if tracing is possible for one or more events but not for others.

6.2 Enabling and Disabling Tracing for Native Record Types

A tool uses the 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 directly 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      */
  _Bool enable,                                    /* enable or disable         */
  uint32_t  flags                                  /* event classes to monitor  */
);
```

The argument `device` is a handle to idenitfy the target device whose performance monitoring may be altered by invoking this function. The boolean `enable` indicates whether recording of events should 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`. Table 2 shows the possible return codes for `ompt_target_set_trace_native`. If a single invocation of `ompt_target_set_trace_ompt` is used to enable/disable more than one kind of monitoring, the return code will be 3 if tracing is possible for one or more kinds of monitoring but not for others.

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. *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) (
  int32_t 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) (
  int32_t device_id,
  const ompt_target_buffer_t *buffer,
  size_t bytes,
  ompt_target_buffer_cursor_t begin,
  _Bool buffer_owned
);
```

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 a *buffer request* callback. The argument `bytes` indicates the full size of the buffer. The argument `begin` is an opaque cursor that indicates the position at the beginning of the first record in the buffer. The argument `buffer_owned` indicates whether or not the data pointed to by buffer was allocated

by a call to the buffer request callback for that device. If multiple devices accumulate trace events into a single buffer (as is the case for NVIDIA's CUPTI API), this callback might be invoked with a pointer to one or more trace records in a shared buffer with `buffer_owned=false`. In this case, the callback may not delete the buffer.

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 not to 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 _Bool ompt_target_start_trace(
  ompt_target_device_t *device,                      /* target device handle       */
  ompt_target_buffer_request_callback_t request,     /* fn to request trace buffer */
  ompt_target_buffer_complete_callback_t complete,   /* fn to return trace buffer  */
  ompt_get_target_info_inquiry_t get_info            /* fn to map to host activity */
);

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

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

Each invocation returns `true` if the command succeeded and `false` otherwise. A call to `ompt_target_stop_trace` also implicitly requests that the device flush any buffers that it owns.

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, returns a cursor for the next record in the specified buffer given a cursor for the current record.

```
OMPT_TARG_API _Bool ompt_target_advance_buffer_cursor(
  ompt_target_buffer_t *buffer,
  size_t size,
  ompt_target_buffer_cursor_t current,
  ompt_target_buffer_cursor_t *next
);
```

It returns `true` if the advance is successful and the returned value `*next` is valid.

The function `ompt_target_buffer_get_record_type` allows a tool to inspect the type of a record to determine whether it is a known type 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_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 a record returned by a prior invocation.

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 a buffer.

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

The pointer returned may point into the trace buffer, or 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. If the function returns a non-NULL result, it will also set `*host_op_id` to identify host-side identifier for the operation associated with the record. A subsequent call to `ompt_target_buffer_get_record_native` may overwrite the contents of the fields in a 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 so that a tool can process native records for a device even if the tool lacks native support for the device.

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

A `ompt_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 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 meaning 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`.

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

6.6 Identifiers in Trace Records

Since analysis of trace records is done asynchronous, a tool cannot store data in an `ompt_data_t` field when the event happens. Therefore, in trace records each OpenMP task region, parallel region and thread has an associated identifier of type `ompt_id_t`. A regions identifier is assigned when the region is created. Identifiers assigned to regions on each device are unique from the time an OpenMP runtime is initialized until it is shut down. Tools should not assume that `ompt_id_t` values are small or densely allocated. The value `ompt_id_none` is reserved to indicate an invalid id.

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 3 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 a tool provides an implementation of `ompt_tool` in the application's address space, 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 return value from a call to a tool-provided implementation of `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,
  unsigned int ompt_version
);
```

The second argument to `ompt_initialize_fn_t` 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 2.

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

Looking up functions in the OMPT API. The first argument to `ompt_initialize_fn_t` 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_data`:

```
ompt_interface_fn_t ompt_get_thread_data_fn = lookup("ompt_get_thread_data");
```

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 by 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
);
```

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 4: Meaning of return codes for `ompt_set_callback`.

The function `ompt_set_callback` may only be called within the implementation of `ompt_initialize_fn_t`. The possible return codes for `ompt_set_callback` and their meaning is shown in Table 4. Registration of supported callbacks may fail if this function is called outside `ompt_initialize_fn_t`. 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 determine 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

The OMPT API provides 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 many times during an execution to monitor only code regions of interest. To support this common usage, the OMPT standard defines four values for `command`:

```
1: start or restart monitoring
2: pause monitoring
3: flush tool buffers
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.

This command may be applied whether monitoring is on or off. 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.

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 omp_state_e {
  /* idle (0..15) */
  omp_state_idle                   = 0x00, /* waiting for work           */

  /* work states (16..31) */
  omp_state_work_serial            = 0x10, /* working outside parallel   */
  omp_state_work_parallel          = 0x11, /* working within parallel    */
  omp_state_work_reduction         = 0x12, /* performing a reduction     */

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

  /* barrier wait states (64..79) */
  omp_state_wait_barrier           = 0x40, /* generic barrier            */
  omp_state_wait_barrier_implicit  = 0x41, /* implicit barrier           */
  omp_state_wait_barrier_explicit  = 0x42, /* explicit barrier           */

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

  /* mutex wait states (96..111) */
  omp_state_wait_mutex             = 0x60, /* waiting for any mutex kind */
  omp_state_wait_lock              = 0x61, /* waiting for lock           */
  omp_state_wait_critical          = 0x62, /* waiting for critical       */
  omp_state_wait_atomic            = 0x63, /* waiting for atomic         */
  omp_state_wait_ordered           = 0x64, /* waiting for ordered        */

  /* target wait states (112..127) */
  omp_state_wait_target            = 0x70, /* waiting for target         */
  omp_state_wait_target_data       = 0x71, /* waiting for target data    */
  omp_state_wait_target_update     = 0x72, /* waiting for target update  */

  /* misc (128..143) */
  omp_state_undefined              = 0x80  /* undefined thread state     */
} omp_state_t;
```

A.2 Runtime Event Callbacks

A tool may indicate interest in receiving notification about certain OpenMP runtime events by registering callbacks. When those events occur during execution, the OpenMP runtime will invoke the registered callback in the appropriate 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 ompt_event_e {
  /*--- Mandatory Events ---*/
  ompt_event_thread_begin             = 1,
  ompt_event_thread_end               = 2,
  ompt_event_parallel_begin           = 3,
  ompt_event_parallel_end             = 4,
  ompt_event_task_create              = 5,
  ompt_event_task_schedule            = 6,
  ompt_event_implicit_task            = 7,
  ompt_event_target                   = 8,
  ompt_event_target_data              = 9,
  ompt_event_target_submit            = 10,
  ompt_event_control                  = 11,
  ompt_event_runtime_shutdown         = 12,

  /*--- Optional Events for Blame Shifting ---*/
  ompt_event_idle                     = 13,
  ompt_event_sync_region_wait         = 14,
  ompt_event_mutex_release            = 15,

  /*--- Optional Events for Instrumentation-based Tools --- */
  ompt_event_task_dependences         = 16,
  ompt_event_task_dependence_pair     = 17,
  ompt_event_worksharing              = 18,
  ompt_event_master                   = 19,
  ompt_event_target_data_map          = 20,
  ompt_event_sync_region              = 21,
  ompt_event_init_lock                = 22,
  ompt_event_destroy_lock             = 23,
  ompt_event_mutex_acquire            = 24,
  ompt_event_mutex_acquired           = 25,
  ompt_event_nested_lock              = 26,
  ompt_event_flush                    = 27
} 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_fn_t                        */

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

typedef uint64_t ompt_id_t;
typedef union ompt_data_u {
  ompt_id_t id;                                 /* integer ID under tool control               */
  void *ptr;                                    /* pointer under tool control                  */
} ompt_data_t;

ompt_data_t ompt_data_none = {.id=0};           /* initial value of ompt_data_t instances      *
                                                 * provided by the runtime                     */

typedef uint64_t ompt_wait_id_t;                /* identifies what a thread is awaiting        */
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_unknown                 = 4
} 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            = 1,
  ompt_sync_region_taskwait           = 2,
  ompt_sync_region_taskgroup          = 3
} ompt_sync_region_kind_t;

typedef enum ompt_target_data_op_e {
  ompt_target_data_alloc              = 1,
  ompt_target_data_transfer_to_dev    = 2,
  ompt_target_data_transfer_from_dev  = 3,
  ompt_target_data_delete             = 4
} ompt_target_data_op_t;

typedef enum ompt_worksharing_type_e {
  ompt_worksharing_loop               = 1,
  ompt_worksharing_sections           = 2,
```

```
  ompt_worksharing_single_executor    = 3,
  ompt_worksharing_single_other       = 4,
  ompt_worksharing_workshare          = 5,
  ompt_worksharing_distribute         = 6
} 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_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_target_type_e {
  ompt_target                         = 1,
  ompt_target_enter_data              = 2,
  ompt_target_exit_data               = 3,
  ompt_target_update                  = 4
} ompt_target_type_t;

typedef enum ompt_invoker_e {
  ompt_invoker_program                = 1,    /* program invokes master task                */
  ompt_invoker_runtime                = 2     /* runtime invokes master task                */
} ompt_invoker_t;

typedef enum ompt_target_map_flag_e {
  ompt_target_map_flag_to             = 1,
  ompt_target_map_flag_from           = 2,
  ompt_target_map_flag_alloc          = 4,
  ompt_target_map_flag_release        = 8,
  ompt_target_map_flag_delete         = 16
} ompt_target_map_flag_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 struct ompt_frame_s {
  void *exit_frame;                         /* runtime frame that reenters user code       */
  void *enter_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_id_none                 0

#define ompt_mutex_kind_unknown      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.

The function signature `ompt_callback_t` defined immediately below is used by the tool callback registration interface as a placeholder for arbitrary callbacks, regardless of a callback's actual type signature. This approach avoids the need for a separate registration routine for each unique callback signature.

```
/* placeholder signature  */
typedef void (*ompt_callback_t) (
  void
);

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

typedef ompt_interface_fn_t (*ompt_function_lookup_t) (
  const char *interface_function_name /* OMPT_API function name to look up      */
);

/* threads */
typedef void (*ompt_thread_begin_callback_t) (
  ompt_thread_type_t thread_type,    /* type of thread              */
  ompt_data_t *thread_data           /* thread data (tool controlled) */
);

typedef void (*ompt_thread_end_callback_t) (
  ompt_data_t *thread_data           /* thread data (tool controlled) */
);

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

/* parallel regions */
typedef void (*ompt_parallel_begin_callback_t) (
  ompt_data_t *parent_task_data,     /* parent task data (tool controlled) */
  const ompt_frame_t *parent_frame,  /* frame data of parent task   */
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  uint32_t requested_team_size,      /* requested number of threads */
  ompt_invoker_t invoker,            /* who invokes master task?    */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_parallel_end_callback_t) (
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  ompt_data_t *task_data,            /* task data (tool controlled) */
  ompt_invoker_t invoker,            /* who invokes master task?    */
  const void *codeptr_ra             /* return address of api call  */
);
```

```
/* task regions */
typedef void (*ompt_task_create_callback_t) (
  ompt_data_t *parent_task_data,     /* parent task data (tool controlled) */
  const ompt_frame_t *parent_frame,  /* frame data for parent task  */
  ompt_data_t *new_task_data,        /* created task's data (tool controlled) */
  ompt_task_type_t type,             /* type of task being created  */
  _Bool has_dependences,             /* task has data dependences   */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_task_dependences_callback_t) (
  ompt_data_t *task_data,            /* task data (tool controlled) */
  const ompt_task_dependence_t *deps,/* vector of task dependences  */
  int ndeps                          /* number of dependences       */
);

typedef void (*ompt_task_dependence_callback_t) (
  ompt_data_t *src_task_data,        /* dependence source task data (tool controlled) */
  ompt_data_t *sink_task_data        /* dependence sink task data (tool controlled) */
);

typedef void (*ompt_task_schedule_callback_t) (
  ompt_data_t *prior_task_data,      /* descheduled task data (tool controlled) */
  _Bool prior_completed,             /* true if prior task completed*/
  ompt_data_t *next_task_data        /* scheduled task data (tool controlled) */
);

typedef void (*ompt_scoped_implicit_callback_t) (
  ompt_scope_endpoint_t endpoint,    /* begin or end                */
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  ompt_data_t *task_data,            /* task data (tool controlled) */
  uint32_t thread_num                /* OMP thread num              */
);

/* task synchronization */
typedef void (*ompt_scoped_sync_region_callback_t) (
  ompt_sync_region_kind_t kind,      /* barrier, taskwait, taskgroup*/
  ompt_scope_endpoint_t endpoint,    /* begin or end                */
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  ompt_data_t *task_data,            /* task data (tool controlled) */
  const void *codeptr_ra             /* return address of api call  */
);

/* mutual exclusion */
typedef void (*ompt_lock_init_callback_t) (
  _Bool is_nest_lock,                /* nested lock or not          */
  ompt_wait_id_t wait_id,            /* wait ID                     */
  uint32_t hint,                     /* OMP lock hint               */
  uint32_t kind,                     /* implementation kind         */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_lock_destroy_callback_t) (
  _Bool is_nest_lock,                /* nested lock or not          */
```

```
  ompt_wait_id_t wait_id,            /* wait ID                      */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_mutex_acquire_callback_t) (
  ompt_mutex_kind_t kind,            /* kind of mutex                */
  uint32_t hint,                     /* based on OMP lock hint       */
  uint32_t impl,                     /* implementation of mutex      */
  ompt_wait_id_t wait_id,            /* ID of mutex being awaited    */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_mutex_callback_t) (
  ompt_mutex_kind_t kind,            /* kind of mutex                */
  ompt_wait_id_t wait_id,            /* ID of mutex being awaited    */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_scoped_nested_lock_callback_t) (
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_wait_id_t wait_id,            /* ID of mutex being awaited    */
  const void *codeptr_ra             /* return address of api call  */
);

/* master */
typedef void (*ompt_scoped_master_callback_t) (
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  ompt_data_t *task_data,            /* task data (tool controlled) */
  const void *codeptr_ra             /* return address of api call  */
);

/* worksharing regions */
typedef void (*ompt_scoped_worksharing_callback_t) (
  ompt_worksharing_type_t wstype,    /* loop, sections, single ...  */
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_data_t *parallel_data,        /* parallel data (tool controlled) */
  ompt_data_t *task_data,            /* task data (tool controlled) */
  const void *codeptr_ra             /* return address of api call  */
);

/* target regions */
typedef void (*ompt_scoped_target_callback_t) (
  int32_t device_id,                 /* ID of the device             */
  ompt_target_type_t kind,           /* target, target data,...      */
  ompt_data_t *task_data,            /* encountering task data (tool controlled) */
  ompt_scope_endpoint_t endpoint,    /* begin or end                 */
  ompt_id_t target_id,               /* ID of the target region      */
  const void *codeptr_ra             /* return address of api call  */
);

typedef void (*ompt_target_data_callback_t) (
  ompt_id_t target_id,               /* ID of the target region      */
  ompt_id_t host_op_id,              /* host side ID for operation  */
```

```
  ompt_target_data_op_t optype,      /* type of operation           */
  void *host_addr,                   /* host address of the data    */
  void *device_addr,                 /* device address of the data  */
  size_t bytes                       /* number of bytes mapped      */
);

typedef void (*ompt_target_data_map_callback_t) (
  ompt_id_t target_id,
  uint32_t nitems,                   /* # of items to be mapped     */
  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         */
);

typedef void (*ompt_target_submit_callback_t) (
  ompt_id_t target_id,               /* ID of the target region     */
  ompt_id_t host_op_id,              /* host side ID for operation  */
  uint32_t requested_num_teams,      /* number of teams requested   */
  uint32_t granted_num_teams         /* number of teams granted     */
);

/* target trace buffer management routines */
typedef void (*ompt_target_buffer_request_callback_t) (
  int32_t device_id,                 /* target device               */
  ompt_target_buffer_t** buffer,     /* host buffer for trace       */
  size_t *bytes                      /* buffer size in bytes        */
);

typedef void (*ompt_target_buffer_complete_callback_t) (
  int32_t device_id,                 /* target device               */
  const ompt_target_buffer_t *buf,   /* target trace records        */
  size_t bytes,                      /* valid bytes in the buffer   */
  ompt_target_buffer_cursor_t begin, /* position of first record    */
  _Bool buffer_owned                 /* true = may delete buffer    */
);

typedef void (*ompt_get_target_info_inquiry_t) (
  ompt_id_t *target_id,              /* target region id            */
  ompt_id_t *host_op_id              /* host side ID for operation  */
);

/* application tool control */
typedef void (*ompt_control_callback_t) (
  uint64_t command,                  /* command of control call     */
  uint64_t modifier,                 /* modifier of control call    */
  const void *codeptr_ra             /* return address of api call  */
);

/* flush */
typedef void (*ompt_flush_callback_t) (
  ompt_data_t *thread_data,          /* thread data (tool controlled) */
  const void *codeptr_ra             /* return address of api call  */
);
```

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_fn_t`, as described in Section 7.2 and Appendix A.7.

```
/* set and get callback functions */
OMPT_API int ompt_set_callback(
  ompt_event_t event,           /* the event of interest                                */
  ompt_callback_t callback      /* function pointer for the callback                    */
);

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

/* enumerate all states used by a runtime */
OMPT_API _Bool ompt_enumerate_states(
  omp_state_t current_state,    /* a valid state in the enumeration                     */
  omp_state_t *next_state,      /* the next state (or none) in the enumeration          */
  const char **next_state_name  /* name of the next state in the enumeration            */
);

/* enumerate all kinds of mutex implementations */
OMPT_API _Bool ompt_enumerate_mutex_kinds(
  uint32_t current_kind,        /* a mutex kind in the enumeration                      */
  uint32_t *next_kind,          /* the next mutex kind (or none) in the enumeration     */
  const char **next_kind_name   /* name of the next mutex in the enumeration            */
);

/* thread inquiry */
OMPT_API ompt_data_t *ompt_get_thread_data(
  void
);

/* state inquiry */
OMPT_API omp_state_t ompt_get_state(
  ompt_wait_id_t *wait_id       /* for wait states: identify what is awaited            */
);

/* parallel region inquiry */
OMPT_API _Bool ompt_get_parallel_info(
  int ancestor_level,           /* how many levels outside the current region           */
  ompt_data_t **parallel_data,  /* return parallel data at specified level (tool controlled) */
  int *team_size                /* return size of the team for the region at ancestor_level */
);

/* task region inquiry */
OMPT_API _Bool ompt_get_task_info(
  int ancestor_level,           /* how many levels outside the current region           */
  ompt_task_type_t *type,       /* return the type of the task                          */
  ompt_data_t **task_data,      /* return task data at specified level (tool controlled) */
  ompt_frame_t **task_frame,    /* return the task_frame of the task                    */
  ompt_data_t **parallel_data,  /* return parallel data at specified level (tool controlled) */
```

```
  uint32_t *thread_num          /* OMP thread num in enclosing parallel region         */
);

/* target region inquiry */
OMPT_API _Bool ompt_get_target_info(
  ompt_id_t *target_id,   /* target region id                                        */
  ompt_id_t *host_op_id   /* host side ID for operation                              */
);

/* target device inquiry */
OMPT_API _Bool ompt_target_get_device_info(
  int32_t device_id,
  const char **type,
  ompt_target_device_t **device,
  ompt_function_lookup_t *lookup,
  const char **documentation
);

OMPT_API int ompt_target_get_device_id(
  void
);

OMPT_TARG_API ompt_target_time_t ompt_target_get_time(
  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                            */
  _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                            */
  _Bool enable,                  /* enable or disable                               */
  uint32_t  flags                /* event classes to monitor                        */
);

OMPT_TARG_API _Bool ompt_target_start_trace(
  ompt_target_device_t *device,                      /* target device handle        */
  ompt_target_buffer_request_callback_t request,     /* fn to request trace buffer  */
  ompt_target_buffer_complete_callback_t complete,   /* fn to return trace buffer   */
  ompt_get_target_info_inquiry_t get_info            /* fn to map to host activity  */
);

OMPT_TARG_API _Bool ompt_target_pause_trace(
  ompt_target_device_t *device,  /* target device handle                            */
  _Bool pause                    /* pause if true; resume if false                  */
```

```
);

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

/* target trace record processing */
OMPT_TARG_API _Bool ompt_target_advance_buffer_cursor(
  ompt_target_buffer_t *buffer,        /* handle for target trace buffer                 */
  size_t size,                         /* the length of valid data in the 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 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_id_t *host_op_id                /* host side ID for operation                     */
);

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_type_e {
  ompt_record_ompt               = 1,
  ompt_record_native             = 2,
  ompt_record_invalid            = 3
} ompt_record_type_t;

typedef enum ompt_record_native_class_e {
  ompt_record_native_class_info  = 1,
  ompt_record_native_class_event = 2
} 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;
  uint64_t hwid;
} ompt_record_native_abstract_t;

/* record types */
typedef struct ompt_record_thread_begin_s {
  ompt_thread_type_t thread_type;    /* type of thread             */
} ompt_record_thread_begin_t;

typedef struct ompt_record_idle_s {
  ompt_scope_endpoint_t endpoint;    /* begin or end               */
} ompt_record_idle_t;

typedef struct ompt_record_parallel_begin_s {
  ompt_id_t parent_task_id;          /* ID of parent task          */
  const ompt_frame_t *parent_frame;  /* frame data of parent task  */
  ompt_id_t parallel_id;             /* ID of parallel region      */
  uint32_t requested_team_size;      /* requested number of threads */
  ompt_invoker_t invoker;            /* who invokes master task?   */
  const void *codeptr_ra;            /* return address of api call */
} ompt_record_parallel_begin_t;

typedef struct ompt_record_parallel_end_s {
  ompt_id_t parallel_id;             /* ID of parallel region      */
  ompt_id_t task_id;                 /* ID of task                 */
  ompt_invoker_t invoker;            /* who invokes master task?   */
  const void *codeptr_ra;            /* return address of api call */
} ompt_record_parallel_end_t;

typedef struct ompt_record_task_create_s {
  ompt_id_t parent_task_id;          /* ID of parent task          */
  const ompt_frame_t *parent_frame;  /* frame data for parent task */
  ompt_id_t new_task_id;             /* ID of created task         */
  ompt_task_type_t type;             /* type of task being created */
  _Bool has_dependences;             /* task has data dependences  */
```

```
  const void *codeptr_ra;           /* return address of api call  */
} ompt_record_task_create_t;

/* note: not task dependences record since it points to data       */
/*        rather than containing it                                */

typedef struct ompt_record_task_dependence_s {
  ompt_id_t src_task_id;            /* ID of dependence source     */
  ompt_id_t sink_task_id;           /* ID of dependence sink       */
} ompt_record_task_dependence_t;

typedef struct ompt_record_task_schedule_s {
  ompt_id_t prior_task_id;          /* ID of descheduled task      */
  _Bool prior_completed;            /* true if prior task completed*/
  ompt_id_t next_task_id;           /* ID of scheduled task        */
} ompt_record_task_schedule_t;

typedef struct ompt_record_scoped_implicit_s {
  ompt_scope_endpoint_t endpoint;   /* begin or end                */
  ompt_id_t parallel_id;            /* ID of parallel region       */
  ompt_id_t task_id;                /* ID of task                  */
  uint32_t thread_num;              /* OMP thread num              */
} ompt_record_scoped_implicit_t;

typedef struct ompt_record_scoped_sync_region_s {
  ompt_sync_region_kind_t kind;     /* barrier, taskwait, taskgroup*/
  ompt_scope_endpoint_t endpoint;   /* begin or end                */
  ompt_id_t parallel_id;            /* ID of parallel region       */
  ompt_id_t  task_id;               /* ID of task                  */
  const void *codeptr_ra;           /* return address of api call  */
} ompt_record_scoped_sync_region_t;

typedef struct ompt_record_lock_init_s {
  _Bool is_nest_lock;               /* nested lock or not          */
  ompt_wait_id_t wait_id,           /* wait ID                     */
  uint32_t hint,                    /* OMP lock hint               */
  uint32_t kind,                    /* implementation kind         */
  const void *codeptr_ra;           /* return address of api call  */
} ompt_record_lock_init_t;

typedef struct ompt_record_lock_destroy_s {
  _Bool is_nest_lock;               /* nested lock or not          */
  ompt_wait_id_t wait_id;           /* ID of mutex being awaited   */
  const void *codeptr_ra;           /* return address of api call  */
} ompt_record_lock_destroy_t;

typedef struct ompt_record_mutex_acquire_s {
  ompt_mutex_kind_t kind;           /* kind of mutex               */
  uint32_t hint;                    /* based on OMP lock hint      */
  uint32_t impl;                    /* implementation of mutex     */
  ompt_wait_id_t wait_id;           /* ID of mutex being awaited   */
  const void *codeptr_ra;           /* return address of api call  */
} ompt_record_mutex_acquire_t;
```

```
typedef struct ompt_record_mutex_s {
  ompt_mutex_kind_t kind;            /* type of mutex               */
  ompt_wait_id_t wait_id;            /* ID of mutex being awaited   */
  const void *codeptr_ra;            /* return address of api call  */
} ompt_record_mutex_t;

typedef struct ompt_record_scoped_nested_lock_s {
  ompt_scope_endpoint_t endpoint;    /* begin or end                */
  ompt_wait_id_t wait_id;            /* ID of mutex being awaited   */
  const void *codeptr_ra;            /* return address of api call  */
} ompt_record_scoped_nested_lock_t;

typedef struct ompt_record_target_data_s {
  ompt_id_t host_op_id;              /* host side ID for operation  */
  ompt_target_data_op_t optype;      /* type of operation           */
  void *host_addr;                   /* host address of the data    */
  void *device_addr;                 /* device address of the data  */
  size_t bytes;                      /* number of bytes mapped      */
  ompt_target_time_t end_time;       /* end time                    */
} ompt_record_target_data_t;

typedef struct ompt_record_target_kernel_s {
  ompt_id_t host_op_id;              /* host side ID for operation  */
  uint32_t granted_num_teams;        /* number of teams granted     */
  ompt_target_time_t end_time;       /* end time                    */
} ompt_record_target_kernel_t;

typedef struct ompt_record_scoped_master_s {
  ompt_scope_endpoint_t endpoint;    /* begin or end                */
  ompt_id_t parallel_id;             /* ID of parallel region       */
  ompt_id_t  task_id;                /* ID of task                  */
  const void *codeptr_ra;            /* return address of api call  */
} ompt_record_scoped_master_t;

typedef struct ompt_record_scoped_worksharing_s {
  ompt_worksharing_type_t wstype;    /* loop, sections, single ...  */
  ompt_scope_endpoint_t endpoint;    /* begin or end                */
  ompt_id_t parallel_id;             /* ID of parallel region       */
  ompt_id_t task_id;                 /* ID of task                  */
  const void *codeptr_ra;            /* return address of api call  */
} ompt_record_scoped_worksharing_t;

typedef struct ompt_record_flush_s {
  void *codeptr_ra;                  /* return address of api call  */
} ompt_record_flush_t;

/* OMPT record */
typedef struct ompt_record_ompt_s {
  ompt_event_t type;                              /* event type          */
  ompt_target_time_t time;                        /* record timestamp    */
  ompt_id_t thread_id;                            /* record's thread ID  */
  ompt_id_t target_id;                            /* host context        */
  union {
    ompt_record_thread_begin_t thread_begin;      /* for thread begin    */
```

```
    ompt_record_idle_t idle;                      /* for idle            */
    ompt_record_parallel_begin_t parallel_begin;  /* for parallel begin  */
    ompt_record_parallel_end_t parallel_end;      /* for parallel end    */
    ompt_record_task_create_t task_create;        /* for task create     */
    ompt_record_task_dependence_t task_dep;       /* for task dependence */
    ompt_record_task_schedule_t task_sched;       /* for task schedule   */
    ompt_record_scoped_implicit_t implicit;       /* for implicit task   */
    ompt_record_scoped_sync_region_t sync_region; /* for sync            */
    ompt_record_target_data_t data;               /* for target data     */
    ompt_record_target_data_map_t data_map;       /* for target data map */
    ompt_record_target_kernel_t kernel;           /* for target kernel   */
    ompt_record_init_lock_t lock_init;            /* for lock init       */
    ompt_record_lock_destroy_t lock_destroy;      /* for lock destroy    */
    ompt_record_mutex_acquire_t mutex_acquire;    /* for mutex acquire   */
    ompt_record_mutex_t mutex;                    /* for mutex           */
    ompt_record_scoped_nested_lock_t nested_lock; /* for nested lock     */
    ompt_record_scoped_master_t master;           /* for master          */
    ompt_record_scoped_worksharing_t worksharing; /* for workshares      */
    ompt_record_flush_t flush;                    /* for flush           */
  } 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

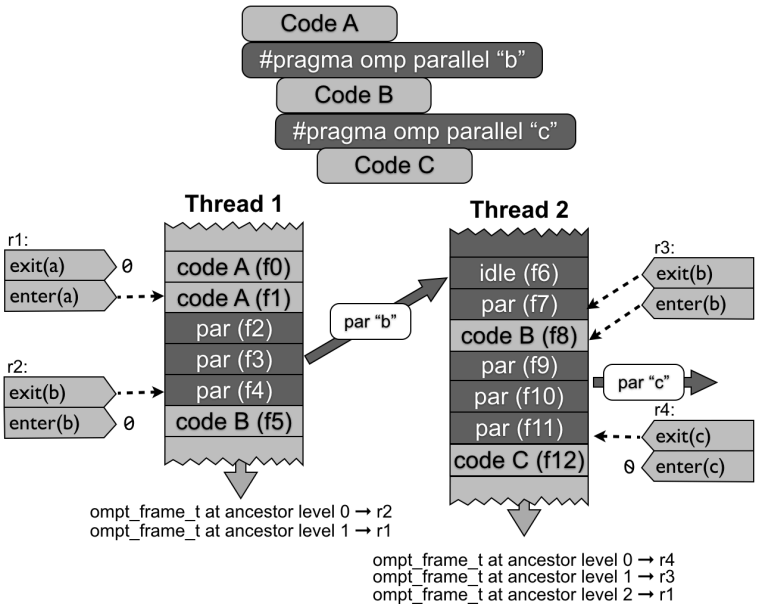

Figure 1: Frame information.

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 `enter_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_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_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 `enter_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_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 Issues

- distribute (fix all of the implications)
- teams?