3.1 UMT and OMPT

UMT versions prior to 2013 uses a Python script as its main program. To study the performance of UMT, we used library preloading to add Rice University's HPCToolkit package for sampling-based performance monitoring into the address space as the Python script for UMT was launched. Python dynamically loads a native shared library `libTeton.so`, which containing the UMT core, into the process address space and invokes an entry point in that library. Eventually, the library code encounters an OpenMP parallel construct and initializes OpenMP. As part of the OMPT design, when an OpenMP is initialized, it must invoke a tool-supplied copy of the function `ompt_initialize`, if one is present.

When HPCToolkit received a callback to `ompt_initialize` to initialize its support for OMPT, we were surprised to find that HPCToolkit's code could not call the OMPT routine `ompt_set_callback`—an entry point point in an OMPT-augmented implementation of Intel's OpenMP runtime that we had developed. Because of the strange visibility rules for dynamically loaded libraries, `ompt_set_callback` isn't visible to our preloaded library even though the OpenMP library is loaded into the address space and is calling `ompt_initialize` in our tool. Since the OpenMP library is loaded by UMT's `libTeton.so`, the global

symbols exported by the OpenMP library are visible only inside `libTeton.so` but not a pre-loaded library, such as HPCToolkit's measurement infrastructure.

This experience left with the question: how do we adjust the design of OMPT so that OMPT tools are insensitive to this symbol visibility problem?

3.2 An Improved Design for OMPT Tool Initialization

The limited visibility of OpenMP global symbols, e.g., `ompt_set_callback`, from a pre-loaded tool library caused us to rethink the design of the OMPT interface for tool initialization. Fortunately, a simple solution enabled us to avoid the problem. Rather than have tools rely on the dynamic linker to resolve symbols for OMPT API functions such as `ompt_set_callback` when they are invoked by a tool, we designed a new interface for the `ompt_initialize` function that enables us to have the OMPT implementation itself resolve symbols directly.

Specifically, we changed the interface to `ompt_initialize` to the following:

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

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

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

Within a tool, one uses `lookup` to obtain function pointers to each OMPT inquiry function. For example, to obtain a function pointer to `ompt_get_thread_id`, one invokes `lookup` as follows:

```
ompt_set_callback_t ompt_set_callback =
        (ompt_set_callback_t) lookup("ompt_set_callback");
```

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

This new design for `ompt_initialize`, motivated by our experiences with UMT, has been accepted by the OpenMP tools committee as part of the emerging OMPT interface.