Providing Observability for OpenMP 3.0 Applications

Yuan Lin[1] and Oleg Mazurov[2]

[1] Nvidia Corp., Santa Clara, CA 95050, USA
yulin@nvidia.com
[2] Sun Microsystems Inc., Menlo Park, CA 94025, USA
oleg.mazurov@sun.com

Abstract. Providing observability for OpenMP applications is a technically challenging task. Most current tools treat OpenMP applications as native multi-threaded applications. They expose too much implementation detail while failing to present useful information at the OpenMP abstraction level. In this paper, we present a rich data model that captures the runtime behavior of OpenMP applications. By carefully designing interactions between all involved components (compiler, OpenMP runtime, collector, and analyzer), we are able to collect all needed information and keep overall runtime overhead and data volume low.

1 Introduction

For any programming environment, offering observability in the runtime behavior of user applications is equally as important as offering schemes that help to create the application. OpenMP provides a set of high level constructs and APIs that aim to simplify the task of writing high performance and portable parallel applications. The nondeterministic nature of concurrent execution of threads, the program transformations performed by the compilers, and the interactions between user applications and runtime libraries makes program observation (e.g., performance profiling and debugging) more important, and at the same time, more difficult to achieve.

For example, a generic performance analysis tool can only provide rudimentary support for OpenMP performance profiling. The tool may show all native threads used in the process - some may map to OpenMP threads and some may be internal service threads used by the OpenMP runtime library itself. The tool may show native callstacks that barely resemble the caller-callee relationship in user program. The tool may not be able to differentiate the user CPU time used for real work from those used for OpenMP synchronization or caused by OpenMP overhead. In OpenMP, an OpenMP thread executes on behalf of an OpenMP task inside an OpenMP parallel region (implicit or explicit) at any particular moment. If the tool has no knowledge of this context, it cannot present information close to the OpenMP execution model, reducing the help it provides in trouble-shooting OpenMP specific performance problems. To a certain extent, the OpenMP execution environment resembles a virtual machine. We

M.S. Müller et al. (Eds.): IWOMP 2009, LNCS 5568, pp. 104–117, 2009.

believe, for general OpenMP programmers, observability should be provided at the OpenMP abstraction level, in addition to the machine level which exposes internals of the virtual machine.

This paper makes the following contributions. First, it presents a data model upon which the runtime behavior of an OpenMP program can be constructed for observation. Second, it presents a way to extend the current OpenMP profiling API to support the new OpenMP tasking feature, and a way to integrate vendor specific extensions. Third, it presents the techniques for generating the user callstack that reflects the program logic rather than implementation details. And last, it presents the techniques for efficiently generating, recording, and reconstructing various tree structures in the data model.

The rest of the paper is organized as follows. Section 2 describes our OpenMP data model. Section 3 describes extensions to the OpenMP profiling API to support the data model. Section 4 presents our techniques in getting the user callstack. Section 5 presents our techniques in getting the thread tree and the task tree related information. Section 6 uses a quicksort program to illustrate the information that can be constructed and presented to users using the data collected by our techniques. Section 7 describes related work. Section 8 concludes our paper.

2 The Data Model

In this section, we describe a rich *data model* that captures the runtime behavior of an OpenMP application. Any event of interest that happened during the execution of an OpenMP application will have its OpenMP context described in the data model.

Before diving into the details of the data model, we define the components involved in the tool chain:

- A *compiler* that processes program source code, translates OpenMP constructs, produces executable code functionally equivalent to the original, and provides information that allows mapping of various program objects in the executable code back to the source code (e.g., user and outlined function naming, instruction to source line mapping).
- An *OpenMP runtime library* that provides thread/task management, scheduling and synchronizations.
- A *collector* that is a runtime component interacting with the program and the OpenMP runtime to obtain and to record information about dynamic program behavior. Low overhead is crucial for this component. Thus recorded data may be different from what finally makes the data model.
- An *analyzer* that works either on-the-fly or post-mortem. The analyzer implements the data model by reconstructing it from data recorded by the collector and provides a means for the user to access that data model through a set of displays or tools with various mechanisms to manipulate data, such as filtering and aggregation.

The callstack is a crucial piece of information for observability. We need a callstack that hides the details of outlined functions from a user, contains only stack

frames from the user's code and maintains natural caller/callee relationships as specified by the logic of the user's program. A set of collected callstacks can be formed into a tree-like data structure with each path from the root representing a particular callstack. We'll refer to that data structure as the dynamic function call graph or function tree.

OpenMP parallel region information is required to understand the parallel behavior of a program. By assuming that serial program execution is represented by the default parallel region with only one thread in the team, we can assert that any program event occurs in some OpenMP thread (i.e., a member of the current parallel region thread team). With nested parallelism and a parent-child relationship between parallel regions we get a tree-like data structure where we can map any event to a node thus specifying all ancestor OpenMP threads and parallel regions up to the default one, which is the root of the tree. We call that data structure the OpenMP thread tree or parallel region tree.

The OpenMP runtime provides a unique ID for each parallel region instance. To collect and to present the entire dynamic parallel region tree would require an enormous amount of data. Two optimizations appear practical: a sampled parallel region tree; and a representation in which all dynamic IDs are mapped to original OpenMP directives in the source code while preserving the dynamic parent-child relationship. This is similar to the dynamic function call graph.

During their lifetime, OpenMP threads transition through various states defined by the OpenMP runtime, such as working in user code, waiting in a barrier or other synchronization objects and performing a reduction operation. Time spent in different states can be aggregated and presented as metrics for various program objects: functions, threads, parallel regions, etc. An OpenMP thread state is thus another important part of the data model.

OpenMP tasks give another perspective to a program's dynamic behavior. By extending the notion of a task for serial execution, parallel regions, worksharing constructs, etc. (and calling such tasks implicit), we can assert that any program event is associated with some task. Task creation defines a parent-child relationship between tasks, which leads us to another tree-like data structure - the OpenMP task tree. Optimizations similar to those suggested for the parallel region tree are also possible.

The OpenMP data model thus consists of the following pieces of information defined for an arbitrary program event: an OpenMP thread state, a node in the parallel region tree, a node in the task tree, and a user callstack. The actual data model may also contain information that is not OpenMP related, such as a time-stamp, a machine callstack, a system thread ID or a physical CPU ID associated with a particular event of interest. We do not discuss how to record and process such information as the techniques are mature and well-known.

3 OpenMP Profiling API

The original OpenMP profiling API[1] was designed to allow for easy extensions, and the introduction of tasks in OpenMP 3.0[2] created an immediate need for such extensions.

To collect information corresponding to the data model described in the previous section, a new set of nine requests is proposed for the common API:

1. depth of the current task tree;
2. task ID of the n-th ancestor task (0 for the current task);
3. source location of the n-th ancestor task;
4. depth of the current parallel region tree;
5. parallel region ID of the n-th ancestor parallel region (0 for the current one);
6. source location of the n-th ancestor parallel region;
7. OpenMP thread ID of the n-th ancestor thread in the parallel region tree path;
8. size of the thread team of the n-th ancestor parallel region;
9. OpenMP state of the calling thread.

Notice that although some of the above information (such as the size of the thread team) is available through standard OpenMP API calls (such as `omp_get_team_size()`), the collector runtime should use the profiling API[1] because it guarantees the above nine pieces of information are provided consistently and atomically in one API call.

Although not used in the work presented in this paper, a set of new events is proposed to cover tasks in OpenMP 3.0: a new task created, task execution begins, task execution suspended, task execution resumed, task execution ends (Appendix A).

Recognizing that some interactions between the collector and the OpenMP runtime can be very specific to a particular implementation, we are suggesting a simple way to add vendor specific requests and events, which would not interfere with possible future extensions of the common API. A vendor willing to implement its own set of extensions should reserve one request number in the common API to avoid possible collision of similar requests from other vendors. This request is issued during the rendezvous to check if that vendor's extensions are supported by the OpenMP runtime and if so, to enable them. All actual extended requests and events are assigned negative values, which will never be used by the common API, and are put in a separate include file. This scheme assumes that no two sets of extensions can be enabled at the same time but it allows both the OpenMP runtime and the collector to support more than one vendor extension. Thus, there is no problem with possible overlap of values or names of actual extension requests and events defined by different vendors.

4 Collecting User Call Stack Information

4.1 The Challenges

OpenMP 3.0 introduces a new tasking feature[1] which makes it easier to write more efficient parallel applications that contain nested parallelism or dynamically

[1] For conciseness, we use the OpenMP task construct to illustrate the challenges and our solution, as it is the most difficult construct to deal with. Similar techniques can be applied to other OpenMP constructs.

generated concurrent jobs. To allow for concurrency, the execution of a task can be deferred and the thread that executes a task may be different from the thread that creates the task. In Fig. 4.1 (a), function `goo()` may be executed by thread 1 while function `bar()` may be executed by thread 2. And function `bar()` may be executed after function `foo()` has returned.

Most OpenMP implementations use the *outlining* technique that generates an *outlined* function that corresponds to the body of many OpenMP constructs and uses a runtime library to schedule the execution of outlined functions among OpenMP threads. Fig. 4.1 (b) illustrates the transformed code. Fig. 4.1 (c) illustrates the interaction between the compiler generated code and the OpenMP runtime library. Notice that function `foo()` now calls an entry point `_mt_TaskFunction_()` in the OpenMP runtime which may asynchronously executes the encountered task on another thread.

Getting the user callstack is not straightforward. In Fig. 4.1, when we inspect the native call stack while the program is executing `bar()`, the native callstack will be very different from the native callstack in code that does not have the OpenMP construct. Fig. 4.1 (d) illustrates the differences. First, the native callstack has frames that are from the OpenMP runtime library. Second, the outlined function is called by a slave thread in a dispatching function inside the OpenMP runtime library. The frames from the root down to the outlined functions are all from the runtime library. Last but not least, function `foo()` may have returned. None of the native callstacks in any of the threads show where the task associated with `__foo_task1()` comes from. All these complications are implementation details that users usually do not care about, have no knowledge of, and are often confused by. To make things worse, the internal implementation scheme may change from one version of implementation to another.

4.2 Scheme Overview

At any moment in an OpenMP application, a thread is executing some OpenMP task if it is not idle waiting for work. Therefore, the user callstack (*UC*) for any event is made of two pieces: *task spawn user callstack (TSUC)* and *local segment (LS)*.

$$UC = TSUC + LS$$

The *TSUC* is the user callstack for the spawn event of the current task. The *LS* is the callstack corresponding to the execution of the outlined task function.

Let's assume for the moment that we know how to get the local segment, then the basic scheme of constructing the user callstacks becomes quite straightforward. When a task is spawned, we get the user callstack corresponding to the spawn event and store it together with the data structure for the task itself. This user callstack, excluding the PC that calls `_mt_TaskFunction_`, is the TSUC for any subsequent event that happened during the execution of the task. Each thread maintains a record of the current task it is executing. When an event happens, we can find the TSUC of the current task by querying the task data structure. We concatenate it with the local segment and get the user callstack.

```
kar()                 | kar()
{                     | {
   foo();             |     foo();
}                     | }
                      |
foo()                 | foo()
{                     | {
    goo();            |     goo();
    #pragma omp task  |     taskinfo = ...
    {                 |     _mt_TaskFunction_(taskinfo, __foo_task1, ...);
       bar();         | }
    }                 |
}                     | __foo_task1(char *arg)
                      | {
                      |     bar();
                      | }
                      |
bar()                 | bar()
{                     | {
   statement 1;       |    statement 1;
}                     | }
```

(a) (b)

```
        User's Code                    .          OpenMP runtime library
   +-------------------------------------------------------------------------
   |  foo()                           .
 T |  {                               .
 h |     goo();                       .
 r |     taskinfo = ...               .
 e |     _mt_TaskFunction_(taskinfo, -----> _mt_TaskFunction_(taskinfo,
 a |                __foo_task1, ...);  .                    mfunc_entry,...)
 d |  }                               .     {    ...
   |                                  .          stored_entry = mfunc_entry;
 1 |                                  .          ...
   |                                  .     }
 ..........................................................................
   |                                  .
 T |                                  .      dispatcher ()
 h |                                  .      {
 r |                                  .           ...
 e |   __foo_task1(char *arg)        <-----       (*stored_entry)(...);
 a |   {                              .            ...
 d |       bar();                     .       }
   |   }                              .
 2 |                                  .
   |                                  .
```

(c)

```
Native Callstack                          User Callstack
================                          ==============
_lwp_start()                              main()
<frames in the OpenMP runtime library>    kar()
dispatcher()                              foo()
__foo_task1()                             bar()
bar()
```

(d)

Fig. 1. (a) original code; (b) compiler transformed code; (c) interaction between compiled code and OpenMP runtime library; (d) native callstack vs user callstack

Notice that the construction of user callstacks is conceptually recursive, and the *TSUC* is always empty for the outer-most task.

In the rest of this section, we will discuss how to get the local segment, and will present a method to get the TSUC more efficiently since computing the user callstack for each task spawn event can be very expensive.

4.3 Getting the Local Segment

The local segment can be constructed by walking up the native callstack when an event happens. Stopping at the first frame from the OpenMP runtime library does not work, because (a) the program may be executing an OpenMP user routine (e.g. `omp_set_num_threads()`); (b) the program may be inside some library that the OpenMP runtime library calls (e.g. `memcpy()` in the standard C library). The key is to tell whether the program is inside an OpenMP user routine, and whether the program is inside an outlined function.

The OpenMP runtime maintains an *in_omp_user_api_state* flag, which is set to 1 whenever the program enters an OpenMP user routine, and is reset to 0 when the program leaves the OpenMP user routine. The OpenMP runtime also maintains a *boundary_stack_pointer*. When the OpenMP runtime library is about to call an outlined function, it records, in the *boundary_stack_pointer*, a stack location in a frame in the call-chain within the OpenMP runtime that will eventually lead to the outlined function. The OpenMP runtime reports the *in_omp_user_api_state* and the *boundary_stack_pointer* to the collector upon request. Section 4.6 describes how the collector uses the two values.

4.4 Getting the Task Spawn User Callstack

The task spawn user callstack (TSUC) is essentially the user callstack (excluding the PC that calls `_mt_TaskFunction_()` (see Fig. fig:exe)) when the task is spawned. Since any task in an invocation of a function will have the same TSUC, we can get the TSUC at the entry of the function instead of computing it every time a task is spawned. This reduces the overhead when multiple tasks are created in one function call, for example inside a loop.

4.5 Pragma PC

In the above description, we assume that, when a profiling event occurs, the thread is either executing some user code or some OpenMP user API calls. However, the thread may also be executing some OpenMP runtime library code (e.g., a thread is enqueuing, dequeuing, or stealing a task). At these moments, the local segment is empty. We are not able to get the leaf PC for the user callstack from the empty segment. The leaf PC that can be used naturally in the user callstack at these moment would be the call instruction to `_mt_TaskFunction_()`. So, we need to get this PC, which we call *pragma_pc*, and report it to the collector, which will use this PC when it finds the local segment is empty.

During the lifetime of a task region, its TSUC and *pragma_pc* will remain constant, while its *boundary_stack_pointer* is set only when the corresponding outlined function is about to be executed.

4.6 Collector Runtime Behavior

We described an idea of storing the task spawn user callstack in the data structure representing a task above in Section 4.2. In our case, a callstack is an array of unprocessed virtual addresses obtained by a stack walking routine implemented in the collector. However, to optimize memory and disk use, the collector implements a mapping scheme using a simple hashing technique. An array of addresses is mapped into a unique 64-bit identifier (UID). The UID can be passed around during program execution, recorded along with other data at an arbitrary time. The mapping scheme guarantees the array of addresses can be reconstructed from the UID during data processing later. The collector records all such mappings on the disk and keeps track of mappings already recorded to reduce data volume. Although unlikely in usual practice, hash collision is still a possibility. Some hash collisions can be detected and reported from different mapping records with the same UID at analysis time. The probability of an undetected hash collision and its cost are considered negligible for the task of statistical performance profiling.

It is trivial to extend to the hashing algorithm so that a new UID can be computed from some previous UID and a segment of addresses. This is essentially the same as appending the segment of address to the array of addresses represented by the previous UID. Therefore, the task spawn user callstack can be represented using a 64-bit UID, which we call *mfunc_start_context_id*. The *mfunc_start_context_id* is obtained by the OpenMP runtime from the collector, stored in a task data structure, and reported back to the collector whenever requested.

We extended the OpenMP profiling API with a new mechanism that allows the OpenMP runtime to issue requests that are carried out by the collector. During the initial rendezvous the collector registers a helper function as a specific event callback. This helper function is called by the OpenMP runtime to obtain a UID for the current user callstack at moments corresponding to task spawning as described above.

When the collector wants to get a user callstack for some program event, it issues a specific OpenMP profiling API request to get the current context, which includes *mfunc_start_context_id*, *boundary_stack_pointer*, *pragma_pc*, and *in_omp_user_api_state*. The collector uses *boundary_stack_pointer*, *pragma_pc*, and *in_omp_user_api_state* to construct the local stack segment. It walks the stack up to the frame pointed to by *boundary_stack_pointer* and collects PC addresses from all stack frames. The collector then checks the collected addresses in the reverse order and if it finds an address from the OpenMP runtime it cuts off the entire tail, possibly leaving the entry address if *in_omp_user_api_state* is set. If the resulting segment is empty, the collector uses *pragma_pc* for the local segment. The collector then computes a UID for the entire user callstack from *mfunc_start_context_id* and the local segment. The computed UID is recorded along with other profiling data for the event.

Because the overall scheme of maintaining user callstacks for OpenMP tasks includes specific mechanisms and interactions, such as UID computation and the

helper mechanism, the request to get the current context is made part of the vendor specific extension to the OpenMP profiling API (Appendix A).

5 Collecting Parallel Region Tree and Task Tree Information

5.1 OpenMP Run Time Part

At any moment, the OpenMP runtime should be able to report, upon query by the collector, the nine pieces of information as described in Section 3. In order to do that, the OpenMP runtime needs to maintain a dynamic tree path during the execution. This is straightforward to implement.

5.2 Collector Part

Instead of collecting and recording information for the entire path in the parallel region tree for each event, the collector tries to reduce overhead and data volume by maintaining the current parallel region ID for each thread in thread local storage (TLS) and recording all necessary information only when it changes. As with callstack UIDs, the collector keeps track of already recorded parallel region IDs. At an event, the collector asks about the current parallel region ID. If the ID is not different from the ID currently stored in TLS, the collector does nothing. Otherwise it records a "thread enters a parallel region" event along with the time-stamp and starts checking if it also needs to record all information about the new parallel region and its ancestors. As multiple threads may almost simultaneously enter a new parallel region, usually only one thread records all information about that parallel region. No synchronization is used between threads to keep track of the recorded status of a parallel region as we only get multiple identical records in the worst case.

At analysis time, we can map any event that is recorded with a time-stamp and a thread ID to an interval determined by "thread enters a parallel region" events, thus obtaining the corresponding parallel region ID. It's guaranteed by the scheme described above that all information about that parallel region and all its ancestors has also been recorded.

The collector uses the same scheme to record task tree information as for recording parallel region tree information.

6 OpenMP Profiling API Examples

We use a quick sort implementation to illustrate how the ideas described above can be presented to the user by a performance analysis tool. All screenshots are obtained from a prototype based on the Sun Studio™Performance Analyzer.

A parallel version of the algorithm using OpenMP is shown in Fig. 6 and the code is pretty straightforward.

```
41.     quick_sort(int lt, int rt, float *data)
42.     {
43.         if ( (rt-lt) < LOW_LIMIT ) {
44.             serial_quick_sort( lt, rt, data );
45.         }
46.         else {
47.             int md = partition( lt, rt, data );
48.             #pragma omp task
49.             quick_sort( lt, md-1, data );
50.             #pragma omp task
51.             quick_sort( md+1, rt, data );
52.         }
53.     }

66.     main(int argc, char* argv[])
67.     {
68.         int n; float *data;
...
98.         #pragma omp parallel
99.         {
100.            #pragma omp single nowait
101.            quick_sort( 0, n-1, data );
102.        }
...
109.    }
```

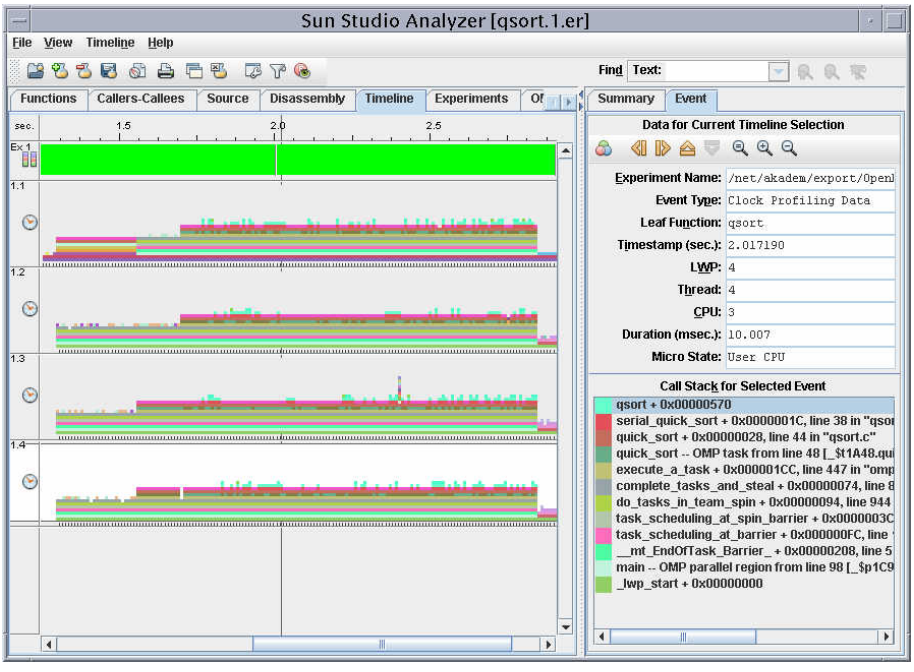
Fig. 2. Parallel quick sort algorithm using OpenMP

An OpenMP unaware tool that is capable of collecting only actual machine callstacks will not show any recursion, because the implementation of OpenMP tasks essentially turns the recursive execution into a work-list based execution. The compiler transformed function of `quick_sort()` contains the initial condition checking, either calls the serial sort function or partitions the specified part of the array and creates two more tasks for sorting both parts of the partition. Instead of recursively calling `quick_sort()`, the program recursively creates tasks. When a task is picked up for execution by a thread, it bears no trace of where it was created and thus the machine callstack has practically the same depth no matter what the logical depth of a particular task is. This behavior can be easily observed in the machine view (Fig. 3 (a)) in Analyzer's Timeline display.

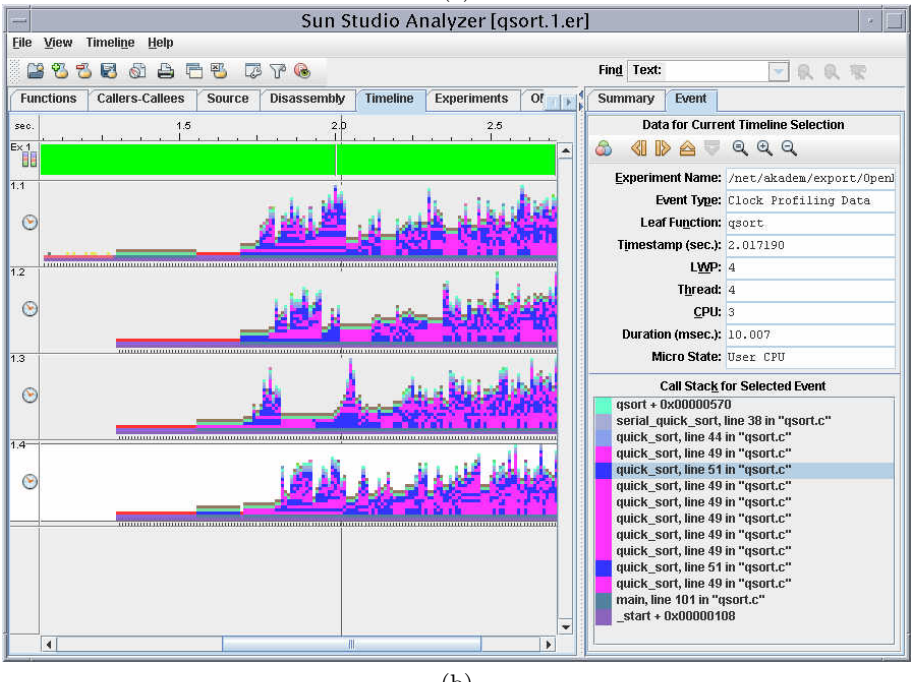
Here, the horizontal axis represents time, and each horizontal bar represents a thread with all collected events shown with their callstacks colored by frame. Selected event details, including the callstack, are shown in the right panel.

In the user view (Fig. 3 (b)), where the logical structure of dynamic program execution is reconstructed, one can see that the recursion pattern with a fluctuating callstack depth is restored.

Knowing the current task ID for each event, we can map it to the original OpenMP construct and compute OpenMP metrics aggregated for those constructs over all events. Two OpenMP metrics, OpenMP Work and OpenMP Wait, are computed based on the OpenMP state recorded for every event. A sorted list of all OpenMP task constructs along with their metrics is presented in the OpenMP tasks display (Fig. 4).



(a)



(b)

Fig. 3. (a) Machine View; (b) User View

```
OMP Work  OMP Wait   Name
 sec.      sec.
-------------------------------------------------------------------------
6.254     1.861      <Total>
2.512     0.010      OpenMP task from quick_sort, line 48 in "qsort.c"
2.342     0.         OpenMP task from quick_sort, line 50 in "qsort.c"
1.141     0.771      OpenMP task 0
0.260     1.081      OpenMP task from main, line 98 in "qsort.c"
```

Fig. 4. OpenMP tasks display

A similar display is provided for all parallel regions. Again, dynamic parallel region IDs are mapped to source and all metrics are computed for the corresponding OpenMP constructs.

7 Related Work

The need for a user level, implementation independent representation of OpenMP program behavior that is consistent with the OpenMP programming model is generally desired and was, in particular, stated in [3] for OpenMP debugging. [3] also emphasized the importance of more detailed views that expose underlying implementation specifics for sophisticated users.

A work towards an open source implementation of the OpenMP profiling API has been reported in [4].

The problem of uniquely identifying OpenMP threads with nested OpenMP parallelism has been approached in [5], where a suggestion, similar to ours, for extension of the standard OpenMP runtime API was made.

User call stacks can also be obtained by tracing function entry and exit events and by maintaining a data structure that allows reconstruction of user call stacks at run time. ompP[6] uses a similar approach to track OpenMP parallel region entry and exit events. It is not clear whether this technique can be extended to deal with tasks without imposing significant overhead, because there usually are significantly more tasks than parallel regions. It is also unclear how this technique would handle untied tasks. Yet another major challenge is dealing with *survived tasks* - a task whose ancestor tasks have finished before the task starts executing.

A general method for efficiently collecting logical call path profiles in multithreaded applications and its implementation for Cilk are described in [7]. The method relies on the availability in the runtime of all pieces of information necessary for logical call path reconstruction at an arbitrary sample point. While that method can certainly cope with work-stealing, as implemented in both Cilk and OpenMP, it's not obvious how it would grapple with survived tasks, which are prohibited by design in Cilk but are allowed in OpenMP.

8 Conclusion

For OpenMP runtime observation tools, such as a debugger and a performance profiling tool, the user model should be intuitive and close to program logic, and

should be presented in terms of high level language constructs used in the program. In this paper, we present a rich data model, which comprises a function tree, a parallel region tree and a task tree, that captures the OpenMP specific runtime behavior. We describe a set of methods that efficiently collect the data for the data model. This work demonstrates that providing high level observability to OpenMP programming and runtime systems, though challenging, is achievable.

Acknowledgements

The authors would like to thank Martin Itzkowitz, Eric Duncan, and Nawal Copty for reviewing the paper. The authors would also like to thank the anonymous reviewers for their many helpful comments and suggestions.

References

1. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: White Paper: An OpenMP Runtime API for Profiling. Tech. Rep., Sun Microsystems, Inc. (2007)
2. OpenMP Architecture Review Board. OpenMP application program interface, version 3.0 (2008), `http://www.openmp.org/mp-documents/spec30.pdf`
3. Cownie, J., DelSignore Jr., J., de Supinski, B., Warren, K.: DMPL: An OpenMP DLL Debugging Interface. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 137–146. Springer, Heidelberg (2003)
4. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Gopalkrishnan, P., Tafti, D.: Towards an Implementation of the OpenMP Collector API. In: Proceedings of the International Conference ParCo (2007)
5. Morris, A., Malony, A., Shende, S.: Supporting Nested OpenMP Parallelism in the TAU Performance System. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 279–288. Springer, Heidelberg (2008)
6. Fuerlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 15–23. Springer, Heidelberg (2008)
7. Nathan, R.: Tallent and John Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, New York (2009)

A OpenMP Profiling API Extensions

```
/* New requests */
typedef enum {
        ...
        OMP_REQ_TASK_NLVLS,     /* depth of the current task tree */
        OMP_REQ_TASK_IDN,       /* task ID of the n-th ancestor task */
        OMP_REQ_TASK_SRCN,      /* source location of the n-th ancestor task */
        OMP_REQ_PREG_NLVLS,     /* depth of the current parallel region tree */
        OMP_REQ_PREG_IDN,       /* ID of the n-th ancestor parallel region */
        OMP_REQ_PREG_SRCN,      /* source location of the n-th ancestor parallel region */
```

```
        OMP_REQ_PREG_THRIDN,    /* thread ID of the n-th ancestor thread */
        OMP_REQ_PREG_TMSZN,     /* thread team size of the n-th ancestor parallel region */
        OMP_REQ_CREATED_TASK,   /* task ID of a newly created task */
} OMP_COLLECTORAPI_REQUEST;


/* New events */
typedef enum {
        ...
        OMP_EVENT_CREATE_TSK,   /* a new task created */
        OMP_EVENT_BEGIN_TSK,    /* task execution begins */
        OMP_EVENT_SUSPEND_TSK,  /* task execution suspended */
        OMP_EVENT_RESUME_TSK,   /* task execution resumed */
        OMP_EVENT_END_TSK       /* task execution ends */
} OMP_COLLECTORAPI_EVENT;

/* New OpenMP thread state */
typedef enum {
        ...
        THR_TSKWT_STATE,        /* waiting in taskwait */
} OMP_COLLECTOR_API_THR_STATE;

/* Reserve request ID for Sun specific extensions */
#define OMP_REQ_SUNEXTENSION  ((OMP_COLLECTORAPI_REQUEST)0x4A415641)

/* Sun specific extensions (from a separate include file) */

#define OMPX_REQ_CONTEXT      ((OMP_COLLECTORAPI_REQUEST)-1)

struct OMPX_request_context {
    int                         size;        /* entry length   */
    OMP_COLLECTORAPI_REQUEST    reqn;        /* request number */
    OMP_COLLECTORAPI_EC         errc;        /* error code     */
    int                         rtsz;        /* return size    */
    uint64_t                    mfunc_start_context_id;
    void                        *boundary_stack_pointer;
    void                        *pragma_pc;
    int                         pragma_pc_state;
    int                         in_omp_user_api;
};

#define OMPX_REGISTER_HELPER  ((OMP_COLLECTORAPI_EVENT)-1)

#define OMPX_HLP_UCTX         ((OMP_COLLECTORAPI_REQUEST)-2)

struct OMPX_helper_uctx {
    int                         size;   /* entry length   */
    OMP_COLLECTORAPI_REQUEST    reqn;   /* request number */
    OMP_COLLECTORAPI_EC         errc;   /* error code     */
    int                         rtsz;   /* return size    */
    void                        *starting_stack_pointer;
    void                        *boundary_stack_pointer;
    uint64_t                    mfunc_start_context_id;
    void                        *pragma_pc;
    uint64_t                    new_context_id; /* return result */
};
```