Open Source Software Support for the OpenMP Runtime API for Profiling

Oscar Hernandez
Ramachandra C Nanjegowda
and Barbara Chapman
Dept. Computer Science
University of Houston
Houston, USA
Email: oscar,rchakena,chapman@cs.uh.edu

Van Bui
Mathematics and
Computer Science Division
Argonne National Laboratories
City, USA
Email: vbui@mcs.anl.gov

Richard Kufrin
NCSA
University of Illinois
at Urbana-Champaign
City, USA
Email: rkufrin@ncsa.uiuc.edu

***Abstract*—OpenMP is a *defacto* standard API for shared memory programming with widespread vendor support and a large user base. The OpenMP Architecture Review Board has sanctioned an interface specification known as the "OpenMP Runtime API for Profiling" to enable tools to collect performance data for OpenMP programs. This paper describes the interface and our experiences implementing it in OpenUH, an open source OpenMP compiler.**

***Keywords*-OpenMP; OpenMP Peformance Tools; OpenMP Runtime API;**

I. Introduction

OpenMP [1] is an API for shared memory parallel programming with directives that are compiled to explicit multi-threaded code. The code produced by the compiler invokes routines in a custom OpenMP runtime library to manage and synchronize threads, and assign work to them. The lack of standards in the runtime layer has hampered the development of third-party tools to support OpenMP application development. The OpenMP Runtime API (ORA) proposed in [4] is an interface specification for profiling/tracing tools for the OpenMP programming model. Prepared by the tools committee of the OpenMP Architecture Review Board (ARB), it is designed to permit a tool, known generically as the *collector*, to gather information about a program's execution from the runtime system in such a manner that neither entity need know any details of the other. Thus it is designed, in particular, to ensure that tools' developers do not need to study the workings of different OpenMP implementations.

ORA [4] is a query and event notification based interface that relies upon *bi-directional* communications between the OpenMP runtime library and performance tools. ORA is independent of the compiler since it resides inside the OpenMP runtime library. It does not require modification of the application's source code since no instrumentation on the user's code is required. Consequently, data collection will interfere much less with compiler analysis/optimizations than corresponding methods, and a more accurate picture of the performance of the application is possible. The design also allows for the collector and OpenMP runtime to evolve independently.

Reference implementations of ORA are sparse since it requires compiler and OpenMP runtime library support. The only known *closed-source* prototype implementation of ORA is provided by the Sun Studio Performance Tools [5]. To the best of our knowledge, the work reported on here is the first *open-source* implementation of ORA. We expect ORA to be adopted by the community as more compiler vendors start to support it.

This paper is organized as follows. In the next section, we discuss related work. We then briefly describe how OpenMP is implemented in order to show the role of the runtime library. This is followed by a discussion of the ORA and our implementation of it. Finally, we indicate the overheads we have measured while using it and discuss future work.

II. Related Work

POMP [8] enables performance tools to detect OpenMP events and was earlier proposed as a standard performance profiling/tracing interface. POMP is independent of the compiler and OpenMP runtime library. It consists of a portable set of instrumentation calls that are designed to be inserted into an application's source code. The instrumentation points measure the time spent inside OpenMP constructs and calls to its user-level library routines. Current tools that support POMP mostly require source code instrumentation (with source code instrumentation tools like Opari), which can interfere significantly with compiler optimizations because the instrumentation calls are interwoven with the application code from the beginning. When used with object code instrumentation, its functionality is restricted to a subset of events. Tools that use POMP are not aware of how OpenMP constructs are translated by the compiler to a specific runtime library.

Other languages have interfaces similar in spirit to ORA. GASP [12] is a profiling interface for global address space programming models such as UPC [2], Co-array Fortran [9], Titanium [14], and SHMEM. GASP is an event-based interface that specifies how the runtime environment communi-

cates with performance tools. The compiler/runtime notifies the tool of a particular action/event through function callbacks to the tool developer's code. GASP does not require any source level instrumentation that would interfere with compiler analysis and optimizations. A potential drawback is the communication overheads from the function callbacks. GASP is currently implemented in the runtime library of Berkeley UPC.

The MPI standard has a performance monitoring interface called PMPI [11] that operates as a software layer between the application source code and the MPI native routines. This interface does not require any source code modifications. PMPI forms a set of wrapper routines for each MPI call such that performance instrumentation calls can occur before and/or after calls to the MPI native routines. From the user's perspective, the MPI interface has not changed at all. The difference is in the implementation of the MPI routine itself, where timer routines are added to collect/store performance data before and after the call to the native MPI event. PMPI has facilitated the development of several performance analysis tools for MPI applications [13], [10], [3].

III. Role of the Runtime in an OpenMP Implementation

Many compilers can translate OpenMP programs into explicit multithreaded code. The typical strategy for doing so involves converting individual OpenMP directives into one or more calls to a custom OpenMP runtime library. The associated program code will also be modified as needed. Clauses that are specified by the application developer along with the directive will influence the manner in which the directive is translated. The runtime library will contain routines to create and manage threads, to assign work to them, and to synchronize their activities. It will also have routines to implement OpenMP's user-level library functions.

For example, the program in Figure 1 shows a simple OpenMP program where the application developer has inserted a directive to specify that a team of threads should perform the reduction in the loop associated with the pragma. A typical strategy for implementing this would package the work of the parallel region into a new procedure, a process known as outlining. Many compilers nest the newly created procedure within the procedure containing the parallel region. Once the threads have been started up, which is accomplished via a call to the corresponding *__ompc_fork()* routine in the runtime library, then each of them will begin to execute this procedure. Here (cf. Fig. 2), the compiler has created a new procedure with the name *__ompdo_main_1*. The threads are started and the routine is passed to them via a call to *__ompc_fork*. In the routine itself, the OpenMP runtime library will pass the thread id of the thread executing the parallel procedure. Then it will invoke another routine in the runtime library (*__ompc_static_init4*) to compute its own set of loop iterations based on the original loop

```
int main(...) {
 int sum =0;
 ...
#pragma omp parallel for reduction(+:sum)
 for  ( i = 0; i < N; i++)
    sum = sum + 1;
 }
 ...
 return 1;
}
```

Figure 1. A sample OpenMP code

```
int main(..) {
 int sum=0;
 ..
__ompc_fork(0, &__ompdo_main1, stack_pointer_of_main1);

/* parallel region function is nested within parent procedure */

 void __ompdo_main1(int __ompv_gtid, /* thread id */
                    void *__ompv_slink_a) {
  int __mplocal_i, __mplocal_sum=0, __mp_lock,
  int __my_lower=0, __my_upper=N-1, my_stride=1;

  __ompc_static_init_4(__ompv_gtid, 2, &my_lower,
                       &my_upper, &my_stride, OMP_STATIC_EVEN, 1);

  for(__mplocal_i = __my_lower; __mplocal_i <= __my_upper;
      __mplocal_i = __mplocal_i + 1) {
    __mplocal_sum = __mplocal_sum + 1;
  }

  __ompc_reduction(__omp_gtid, & __mplock);
  sum = sum + __mplocal_sum;
  __ompc_end_reduction(__omp_gtid, & __mplock);

  __ompc_ibarrier();
  return;
}
 return 1;
} /* end of main */
```

Figure 2. Compiler translation of the sample OpenMP code of Fig. 1 into explicitly multithreaded code

bounds of the parallel loop. It is now ready to perform its portion of the work, in this case a reduction operation. The compiler has modified the loop header accordingly. Finally, the threads enter a critical region bounded by the OpenMP runtime calls *__ompc_reduction* and *__ompc_end_reduction* to update the shared reduction variable *sum* with the local results *__mplocal_sum*. They synchronize at a barrier before the region terminates.

We can use these OpenMP runtime calls to implicitly capture OpenMP performance events and states, such as when a thread performs a fork/join operation and goes from a serial state to another state (i.e. parallel overhead state or parallel work state), enters/exits a barrier or starts to compute a reduction. All these states and events can be captured just by adapting the OpenMP runtime calls, without modifying the OpenMP translation of the original procedure in any way. The ORA interface (which we will describe in the next section) provides an API to query the OpenMP runtime for thread states and event notifications via callback functions.

IV. An Implementation of the OpenMP Profiling API

The ORA interface [4] consists of a single routine that takes the form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array that can be used by a collector to pass one or more requests for information from the runtime. The collector requests notification of a specific event by passing the name of the event to monitor as well as a callback routine to be invoked by the OpenMP runtime each time the event occurs.

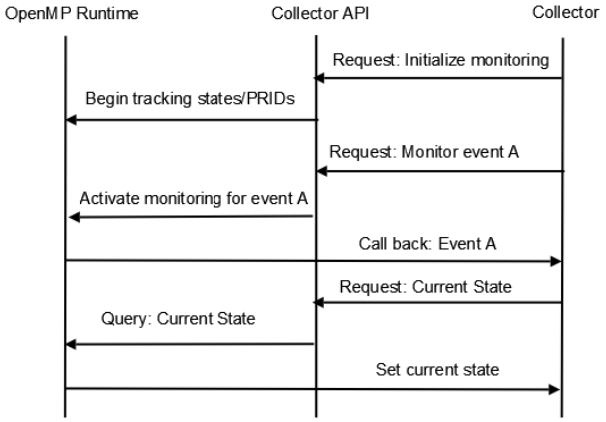

Figure 3. Example of a sequence of requests made by collector to OpenMP runtime.

Since the OpenMP runtime and the collector tools remain fully independent of one another, it is essential that a mechanism be provided that allows each of them to interact while not compromising their ability to operate separately and individually. This is accomplished by having the OpenMP runtime implement a single API function *__omp_collector_api* and export its symbol in the OpenMP runtime library. The collector may then query the dynamic linker to determine whether the symbol is present. If it is, then it may initiate communications with the runtime and begin to make queries of and send requests to the OpenMP runtime using the interface. The OpenMP runtime will then start keeping track of thread states and triggering event notifications. Figure 3 depicts a possible sequence of interactions between the collector and OpenMP runtime.

The thread states that the OpenMP runtime should distinguish include when a thread is doing useful work or executing OpenMP overheads (preparing to fork, or computing scheduling strategies for threads). Also it should distinguish states when a thread is idle or executing an explicit/implicit barrier, performing a reduction, or waiting on locks, critical and ordered regions, or performing an atomic update. When the collector makes a request for notification of a specified event(s), the OpenMP runtime will activate monitoring for this event inside its environment. The collector may also make requests to pause, resume, or stop event generation, each of which must be supported in the runtime. The collector interface specification requires that the OpenMP runtime provide support for notification of fork and join events and specifies support for the other events as optional to support tracing. These optional events include begin/end of explicit and implicit barriers, begin/end of waits for locks, critical regions, ordered sections, atomic operations, begin/end of master, single regions and when slave threads start and stop being idle (in serial regions between parallel regions).

ORA relies on retrieving the program callstack and different IDs (the parallel region id and its parent, etc) to provide the mapping between a given event/state and the source code. This is usually done at the join event to exclude small parallel regions where the collector tool did not gather any information. Reconstructing the callstack to provide a user view of the program is done offline after the application finishes. Some issues to consider regarding ORA's approach to performance measurement includes the overhead associated with activating/deactivating event generation and overheads from invoking the callback routines. Overheads can be minimized by being selective with respect to the events to turn on/off and to the points where the callstack is retrieved (we want to avoid doing so for insignificant events and small parallel regions).

A. OpenMP Collector API support in the Runtime Library

We implemented the OpenMP Collector API in the OpenUH compiler. OpenUH [7], a branch of the Open64 compiler suite, is an optimizing and portable open-source OpenMP compiler for C/C++ and Fortran 95 programs. The suite is based on SGI's Pro64 compiler which has been provided to the community as open source. It targets the Itanium, Opteron, and x86 platforms, for which object code is produced, and may be used as a source-to-source compiler for other architectures using the intermediate representation (IR)- to-source tools. At the time of writing, OpenMP 2.5 is supported along with a partial implementation of OpenMP 3.0; the OpenMP runtime library is open source.

To implement the OpenMP Collector API we had to modify the OpenMP runtime library and the OpenMP translation in the OpenUH compiler. Several extensions are required in the OpenMP runtime in order to support the OpenMP Collector API specification. The OpenMP runtime needs to (1) be able to initiate/pause/resume/stop event generation, (2) respond to queries for the ID of the current/parent parallel region and wait ids, and (3) respond to queries for the current state of the calling thread.

B. Initialization of the OpenMP Collector API

When a collector tool requests the initialization of the OpenMP Collector API, it sends a message request of type *OMP_REQ_-START* to inform the OpenMP runtime that it should start keeping track of thread states, initialize the necessary storage classes (queues) to process requests, initial-

ize all tables that contain callback information/notifications (function pointers), and start keeping track of different IDs (e.g., parallel region IDs, parent parallel region IDs). A thread-safe boolean global variable is used to indicate if the Collector API has been initialized. Any thread can modify this variable to true after the initialized request or to false when a thread requests a stop via *OMP_REQ_STOP*. If two requests for initialization are made without a stop request in-between, an "out of sync" error code is returned. After ORA has been initialized, future requests to the API are pushed onto a queue associated with a thread. In this manner, we were able to avoid the contention otherwise incurred if a single global queue processed requests.

C. Implementing Events and States

When a thread sends a request to register a given event, the event type *OMP_COLLECTORAPI_REQUEST* and a callback function pointer is passed as an argument to the API call in the runtime. This function pointer is stored in a table that contains the event callbacks shared by all the threads. Each table entry has a lock associated with it to avoid data races when multiple threads try to register the same event with different callbacks. This implementation assumes that all threads will share the same set of callback functions for the registered events. It also assumes that the registration of events and callbacks will not be frequent and will mostly occur at program start. Whether these assumptions hold will depend on the nature of the collector tool and how selective it is with regard to registering/unregistering events.

The events and states in our runtime are implemented by inserting the function *__ompc_event((event_name)* and *__ompc_set_state(state_name)* at different locations in the OpenMP runtime. The state values are stored in a field of the OpenMP thread descriptor, a data structure that is kept within the runtime to manage OpenMP threads. Just before an OpenMP thread is created, in our case via *pthread_create()*, its thread descriptor is set up. The master thread is the only thread that can run in parallel or serial mode and because of that it has two thread descriptors. This duplicate data structure was needed because it is possible for a tool to initialize the collector API before the OpenMP runtime library is initialized.

The ORA white paper [4] indicates that is optional for the OpenMP runtime library to keep track of thread states even before or after the Collector API has been initialized. We decided to always keep track of the states once OpenMP threads are created and the OpenMP runtime is initialized. The reason behind this is that we want to avoid the use of conditional statements throughout the OpenMP runtime library to check if the OpenMP Collector API is initialized or is paused, which is not efficient if a program executes without using the OpenMP collector API. Keeping track of the thread states is an inexpensive operation which consists of performing one assignment operation per state to update the private thread descriptor that is local to a thread.

When a thread encounters a point within the OpenMP runtime library that corresponds to an event, it will invoke the callback function related to it: this only occurs if the event has been registered (so there is a callback function associated with it) and the Collector API has been initialized and is not in paused mode. When a thread reaches an event point, the function *__ompc_event((OMP_COLLECTORAPI_EVENT) e)* will execute and use the event type to access the callback table. If the event has been registered, the callback table will contain a function pointer for the corresponding callback; it will contain the value NULL for unregistered events. The ordering of the checks is important to avoid unnecessary checking if no callback has been registered for an event (which is possible if the OpenMP Collector API has not been inizialized). If all conditions are true, then the thread will execute the callback and the event type is passed as an argument to the callback function.

1) Fork and join events and their states: The serial state *THR_OVHD_STATE* is the first state that we keep track of (even if the OpenMP Collector API hasn't been initialized) and is only applicable to the master thread before and after it executes a non-nested parallel region. The fork event is implemented by a call to the function *__ompc_event* with the actual argument of *OMP_EVENT_FORK*, just before the call *pthread_create()* is executed in our implementation. This occurs when the OpenMP runtime library encounters the first parallel region and needs to initialize and create threads and whenever it needs to create more OpenMP threads. As soon as the threads are created, they are set to be in the *THR_IDLE_STATE* and the event *OMP_EVENT_THR_BEGIN_IDLE* triggers a callback associated with that event. In situations where the number of threads executing parallel regions in an OpenMP program changes dynamically, subsequent fork events will be triggered before the call to *pthread_create()* in order to add more threads to the existing ones as required. In our OpenMP implementation, all the threads survive (and are sleeping) in between non-nested parallel regions. Since conceptually in the user model of OpenMP there is a fork operation at the begining of each parallel region (even if the OpenMP runtime does not create new threads), the fork event is triggered while the master thread updates the slave threads' thread descriptors with the newly created outlined OpenMP procedure (see Fig. 2, just before the slave threads begin to execute the parallel region. During this process, the master thread is considered to be in the overhead state. Our compiler currently serializes nested parallel regions and because of this, we do not trigger a fork event for nested parallel regions. This will change in future releases of the compiler and a fork event will be generated whenever we create a nested parallel region and the corresponding

OpenMP threads. In the case of a join operation, the *OMP_EVENT_JOIN* is triggered and the state of the master thread is set to *THR_OVHD_STATE* as soon as it leaves the implicit barrier at the end of the parallel region. The fork and join event callback are only invoked by the master thread of any parallel region.

2) Barrier events: The states and events related to barriers are implemented by setting the thread state variable to *THR_EBAR_STATE* or *THR_IBAR_STATE* (for explicit or implicit barriers). Then, when a thread enters and exits a barrier the threads triggers the events *OMP_EVENT_THR_BEGIN_EBAR* and *OMP_EVENT_THR_END_EBAR* when they exist the barrier. The implementation of these states and events proved to be straightforward, since they were simply inserted into the OpenMP runtime calls that implement a barrier. However, initially our runtime was not able to distinguish whether a barrier was implicit or explicit because it used the same runtime call for both cases. Indeed, the functionality is the same. As a result, we had to change the way our compiler translated OpenMP barriers so that different runtime calls were generated according to whether it was dealing with an implicit or an explicit barrier (see Figure 2 where it has inserted a *__ompc_ibarrier* call. Each thread keeps track of its own implicit or explicit barrier ID, which is incremented each time a thread enters a barrier.

3) Lock wait events and state: Our implemention of OpenMP locks is based on the Pthreads lock implementation. The events *OMP_EVENT_THR_BEGIN_LKWT*, *OMP_EVENT_THR_END_LKWT* were implemented by modifying the implementation of our OpenMP lock runtime call. The state was implementing by setting the the state *THR_LKWT_STATE*. Since these events and state are triggered only when a thread is waiting for a lock, we added the function *pthread_try_lock()* to capture an individual thread's behavior and check whether the lock is available. If it is available, then the thread acquires the lock and continues its execution. If the lock is busy, then we trigger the wait lock state and corresponding event. Also, each thread increments a lock wait ID. The same procedure is applied for nested locks. There are several places within our OpenMP runtime library where implicit locks are used; however we trigger this state and the events only for user-defined locks (i.e. locks that were specified by the user in the source code). Since our compiler initially used the same OpenMP runtime call to implement all locks, we modified it to provide separate implementations for each case.

4) Critical region wait states and events: A similar approach is used to implement the state and events corresponding to a critical region. Our critical region runtime call is implemented by the insertion of a compiler-generated lock that is used to generate a mutually exclusive region (see Figure 2 for the compiler-generated locks in the reduction operation, which are generated in the same way as for critical regions). We implemented the state and events *OMP_EVENT_THR_BEGIN_CTWT*, *OMP_EVENT_THR_END_CTWT* and *THR_CTWT_STATE* within the OpenMP runtime calls for critical regions and when the threads wait to acquire this automatically generated lock. A critical region wait ID is maintained and incremented each time a thread waits to acquire the lock inside a critical region. Each thread keeps track of its own wait IDs.

5) Reduction state: Initially our compiler implemented the reduction operation (at the end of a parallel do or for) by translating it so that it relied upon a critical region to perform the thread updates of the shared reduction value that is the last part of the operation. We therefore modified the OpenMP translation to distinguish between a reduction operation and critical region runtime calls by inserting a special OpenMP runtime call for reductions (see Figure 2). Whenever a thread enters a reduction operation, it automatically sets its state to *THR_REDUC_STATE*.

6) Master and single begin/end events: We needed to change the translation of the OpenMP runtime in order to be able to handle the begin/end master events. Our modified implementation inserts two runtime calls (previouly there was only a conditional with a runtime call) at the beginning and end of the translated *master* construct, respectively. The purpose of these runtime calls is to capture both the events that occur when the master thread enters or leaves the master region. A similar approach was used for *single* regions. The extra runtime call at the end of the translation of the single construct ensures that the single exit event is captured. According to the OpenMP Collector API white paper, the thread's state inside a single or master construct is undefined because these worksharing constructs allow nesting with other OpenMP constructs. However, as a default we set the thread state to *THR_WORK_STATE* for those threads executing them.

7) Atomic wait operations: The events and state for atomic waits were not implemented in our compiler. The reason behind this decision is that triggering these events and states will inevitably incur large overheads. Implementing these events also adversely affects the translation of the OpenMP atomic operation. In OpenUH, atomic operations are translated to intrinsic synchronization operations that are not part of the OpenMP runtime library. However it might be possible to implement this event if we implement a wrapper function against a native implementation or if we use the assembler to modify this.

D. Get state request for the Collector API

The colletor tool can request the state of a thread at any given point of the program execution. We made sure that this type of request could be requested at any given point during the execution of the program. To guarantee that a thread will always have a state associate with it, (even before a thread

is created in the case of the slave threads), this data structure descriptor is initialized to *THR_OVDH_STATE* state to reflect the slave threads are in the process of being created. This would guarantee that that any OpenMP thread will have a state associate with it and will always return a correct value. Also when a state is requested for a thread outside of the parallel region, it will return the *THR_IDLE_STATE* state or *THR_SERIAL_STATE*. Since some states have a wait ID associated with them, we set the corresponding values of these wait IDs as set in the thread descriptor data structures (the values of the barrier ID, or lock wait ID, etc). For example we return the value of a barrier ID or lock ID after the event type in the mem section of the OpenMP collector API request.

E. Parallel Region IDs and Parent Region IDs.

Since a team of threads will execute a parallel region and there is a one-to-one mapping, we added an OpenMP region ID and parent region ID field as a part of the thread team data structure descriptor. Each time a team of threads executes a parallel region, this current and parallel region ID is updated. In the case of nested parallelism, we don't keep track of these IDs because our compiler currently serializes them. However in future releases of the compiler, we will update the Current Parallel Region and Parent Regions ID for the team of threads executing the nested parallel region. In the case of a non-nested parent parallel region ID, its parent region ID will always be zero. In the case of a nested parallel region, it will return the current parallel region ID of the parent team that spawned the new team of threads. We implemented this collector API request so that it can be executed asynchronously and so that we can make this request at any given point during the execution of the program. When a thread is outside a parallel region, it will return an error code indicating a request out of sequence and an ID with the value of zero. This is also the case when a thread goes into the serial state or idle state.

F. Constructing the User-Model Callstack Profile

PerfSuite is an open source software package for application performance analysis that interfaces to the user through command-line tools that can be used with unmodified applications or through a compact API that allows more flexibility in selective monitoring of portions of an application [6]. With the advent of the ORA, we extended the core PerfSuite libraries used within the OpenUH runtime to expose additional contextual information in addition to the raw performance data collected within parallel regions. The extensions support reconstruction of the user model callstack, which is necessary since performance data is typically collected and coupled with the implementation model callstack. These extensions were implemented in an auxiliary library called *libpsx* and provide the following additional capabilities:

- Call-stack retrieval, using the open source library *libunwind*[1]. New API entry points, callable by the collector, provide instruction pointer values for each stack frame at the point of inquiry, allowing reconstruction of the call graph.
- Mapping of instruction pointer values to source code location, using the Binary File Descriptor (BFD) API that is contained in the GNU "binutils" package[2]. Combined with the call-stack retrieval extensions, this allows the collector to assemble information meaningful in the context of the user's source code.

In addition to the ORA implementation provided by OpenUH, these extensions to PerfSuite enable tool developers to design tools that can collect performance data and map this data back to events that make up the user model of OpenMP.

V. Experimental Results

We developed a prototype performance measurement tool to estimate the overheads that can incur from data collection. The performance tool is based on the extensions made to PerfSuite's [6] performance measurement libraries that were discussed in Section IV-F and also on the OpenUH compiler's implementation of ORA. The tool is a shared object that is LD_PRELOAD'ed to the target's address space. It includes an **init** section that queries the runtime linker for the presence of the OpenMP API symbol. If the symbol is present, the tool initiates a **start** request and registers for the *fork*, *join*, and *implicit barrier* events. The callback routine that is invoked each time a registered event occurs at runtime stores a sample of a hardware-based time counter. Furthermore, to estimate the potential overheads from callstack retrieval, the tool also records the current implementation-model callstack for each join event.

A. EPCC Benchmark

We evaluated our approach first by measuring the overheads for the different events/states when ORA is enabled. To do this we used the EPCC Microbenchmarks on an Altix System 3700 and used up to 32 processors. For each event we collect, the framepointer allows us to map the event to the source code. The results from this event can be found in Figure 4. Figure 4 indicates percentage increase in the overheads of different OpenMP directives for 4, 8, 16 and 32 threads for EPCC synch benchmarks (results from the 2 threaded runs are not included because of its low execution time). These benchmarks consists of several instances of parallel region, parallel for, and reduction directives (about 20000 each). These directives incur an overhead of $\tilde{5}\%$ in several instances. The other directives are used a very few number of times and the percentage increase in overhead is

[1] http://www.nongnu.org/libunwind

[2] http://www.gnu.org/software/binutils

less than 5%. A few outlier cases (lock and atomic) have very low execution time, hence the overhead may seem higher (see Figure 4).

B. NAS Parallel Benchmark

Next we evaluated our approach by running experiments on an Intel Xeon workstation with dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 16 GB DDR2 FBDIMM RAM, running Ubuntu 8.04. We ran the NPB3.2-OMP and the NPB3.2-MZ-MPI benchmarks in the experiments. We compiled the benchmarks with the OpenUH compiler, enabling both OpenMP and -O3 level compiler optimizations. For the NPB3.2-OMP benchmarks, we collected data using 1, 2, 4, and 8 threads. On the NPB3.2-MZ-MPI benchmarks, we varied both the process and thread counts, respectively, as follows: 1 X 8, 2 X 4, 4 X 2, and 8 X 1, where the first number indicates the number of processes and the second the number of threads. For both benchmark sets, we used problem size Class B. Figures 5 and 6 graphically depict the overheads from collecting performance data with our collector tool for both benchmark sets.

The results intuitively indicate that a higher number of parallel region calls will result in more overheads, which is not surprising. Figure 5 shows that LU-HP incurs the highest overheads with as much as 6% on eight threads. Table I also indicates that, of the OpenMP benchmarks, LU-HP also has the highest number of calls to a parallel region, with close to 300,000 invocations. For the MPI/OpenMP hybrid benchmarks, SP-MZ incurred the highest overheads (16%) with over 400,000 parallel region invocations for the 1 process X 8 thread case (see Figure 6 and Table II). The 2 X 4 case for SP-MZ follows with close to 8% overheads; it makes about 200,000 parallel region invocations (see Figure 6 and Table II). The overhead measurments for the majority of the other benchmarks in the experiments are less then 5% (see Figures 5 and 6). A few outlier cases, where we observed overhead values of less then 1%, are listed as zero overhead in Figures 5 and 6. The standard deviation values across multiple runs across all benchmarks are less then 2 secs.

In order to understand in more detail where the overheads were coming from, we ran additional experiments for the two benchmarks exhibiting the highest overheads. We ran LU-HP and SP-MZ disabling and enabling the performance data collection. As a result, we were able to measure the costs associated with communications between the OpenMP runtime and the collector utility and overheads associated with performance measurement/storage. We ran SP-MZ for the 4 threads X 1 process case and LU-HP on 4 threads. For LU-HP, the results indicate that 81.22% of the overheads can be attributed to performance measurement/storage. In the case of SP-MZ, 99.35% of the overheads came from performance measurement/storage. For both benchmarks,

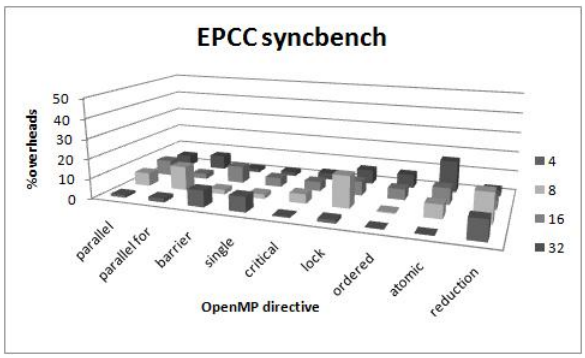

Figure 4. Overhead measurements for EPCC benchmarks.

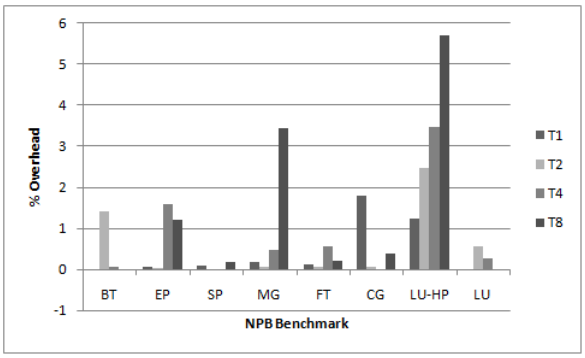

Figure 5. Overhead measurements for NPB3.2-OMP benchmarks.

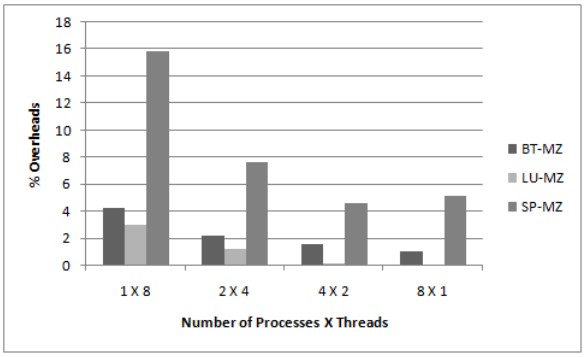

Figure 6. Overhead measurements for NPB3.2-MZ-MPI benchmarks.

the overheads from issuing callbacks and communications between the OpenMP runtime and the collector were significantly less then overheads caused by performance measurement. The results suggest that efforts for reducing overheads should focus on optimizing the measurement/storage phases of performance tool development.

VI. Conclusion

In this paper, we have described the "OpenMP Runtime API for Profiling" (ORA) and our experiences implementing it for an open-source portable OpenMP compiler. Further-

Table I
NUMBER OF PARALLEL REGIONS FOR THE NPB3.2-OMP BENCHMARKS.

Benchmark	# parallel regions	# region calls
BT	11	1014
EP	3	3
SP	14	3618
MG	10	1281
FT	9	112
CG	15	2212
LU-HP	**16**	**298959**
LU	9	518

Table II
NUMBER OF PARALLEL REGION CALLS FOR THE NPB3.2-MZ-MPI BENCHMARKS (PROCESS X THREAD).

Benchmark	1 X 8	2 X 4	4 X 2	8 X 1
BT	167616	83808	41904	20952
LU	40353	20177	10089	5045
SP	**436672**	**218336**	**109168**	**54584**

more, we have implemented tool support for reconstructing the user-level execution behavior of OpenMP. Both of these developments are important facilitators for the design and implementation of OpenMP performance tools. From our experiments, we found that overheads must be carefully controlled in both the OpenMP runtime and in the collection of the measured data in order to obtain the most accurate results. We also found that using both our implementation of ORA and added support for accessing the runtime callstack, we were able to rapidly prototype a tool capable of generating OpenMP specific performance metrics with minimal overheads in most cases.

This implementation of ORA currently supports OpenMP 2.5. More work will be needed to extend the interface to handle the constructs in the recent OpenMP 3.0 standard. Moreover, the interface provides little support for important work-sharing constructs like parallel loops and for relating them to their corresponding barrier events/states. To control the runtime overheads, tools can reduce the number of times data is collected by distinguishing between either the same parallel region or the calling context for a parallel region. We intend to provide additional software support and will design efficient algorithms in our tools to properly address the issues outlined above.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under contract CCF-0702775 and by the Department of Energy under contract DE-FC03-01ER25502. This work was also supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. Van Bui is also affiliated with the Computer Science Department at the University of Houston. The authors thank Boyana Norris and Lois Curfman McInnes of Argonne National Laboratory for providing the system used in the experimental portion of this paper.

REFERENCES

[1] DAGUM, L., AND MENON, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (1998), 46–55.

[2] EL-GHAZAWI, T., CARLSON, W., STERLING, T., AND YELICK, K. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, May 2005.

[3] ITZKOWITZ, M. The sun studio performance tools. Tech. rep., Sun Microsystems Inc., November 2005.

[4] ITZKOWITZ, M., MAZUROV, O., COPTY, N., AND LIN, Y. White paper: An openmp runtime api for profiling. Tech. rep., Sun Microsystems, Inc., 2007.

[5] JOST, G., MAZUROV, O., AND AN MEY, D. Adding new dimensions to performance analysis through user-defined objects. In *IWOMP* (June 2006).

[6] KUFRIN, R. Perfsuite: An accessible, open source performance analysis environment for linux. In *6th International Conference on Linux Clusters: The HPC Revolution 2005* (April 2005).

[7] LIAO, C., HERNANDEZ, O., CHAPMAN, B., CHEN, W., AND ZHENG, W. Openuh: An optimizing, portable openmp compiler. In *12th Workshop on Compilers for Parallel Computers* (January 2006).

[8] MOHR, B., MALONY, A., HOPPE, H., SCHLIMBACH, F., HAAB, G., HOEFLINGER, J., AND SHAH, S. A performance monitoring interface for openmp. In *Proceedings of the 4th European Workshop on OpenMP* (September 2002).

[9] NUMRICH, R. W., AND REID, J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum 17*, 2 (1998), 1–31.

[10] PALLAS GMBH. *Vampirtrace 2.0 Installation and User's Guide*, November 1999.

[11] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[12] SU, H., BONACHEA, D., LEKO, A., SHERBURNE, H., III, M. B., AND GEORGE, A. Gasp! a standardized performance analysis tool interface for global address space programming models. Tech. Rep. LBNL-61659, Lawrence Berkeley National Lab, September 2006.

[13] WOLF, F., AND MOHR, B. Automatic performance analysis of hybrid mpi/openm p applications. *Journal of Systems Architecture: the EUROMICRO Journal 49*, 10-11 (2003), 421–439.

[14] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (New York, NY 10036, USA, 1998), ACM, Ed., ACM Press.