OpenMP Tools Interface: Synchronization Information for Data Race Detection

Joachim Protze[1,2], Jonas Hahnfeld[1,2], Dong H. Ahn[3(✉)], Martin Schulz[3], and Matthias S. Müller[1,2]

[1] RWTH Aachen University, 52056 Aachen, Germany
{protze,hahnfeld,mueller}@itc.rwth-aachen.de
[2] JARA – High-Performance Computing, 52062 Aachen, Germany
[3] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{ahn1,schulzm}@llnl.gov

Abstract. When it comes to data race detection, complete information about synchronization, concurrency and memory accesses is needed. This information might be gathered at various levels of abstraction. For best results regarding accuracy this information should be collected at the abstraction level of the parallel programming paradigm. With the latest preview of the OpenMP specification, a tools interface (OMPT) was added to OpenMP. In this paper we discuss whether the synchronization information provided by OMPT is sufficient to apply accurate data race analysis for OpenMP applications. We further present some implementation details and results for our data race detection tool called Archer which derives the synchronization information from OMPT.

1 Introduction

OpenMP is the de facto standard for parallel programming on shared memory machines. It is also becoming increasingly popular on extreme-scale systems as it offers a portable way to harness the growing degree of parallelism available on each node. However, porting large HPC applications to OpenMP often introduces subtle errors. Of these, data races are particularly egregious, as well as challenging to identify. Data races may remain undetected during testing, but nevertheless manifest during production runs by often resulting in confusing (and/or non-reproducible) executions that the programmer wastes considerable amounts of time debugging. In extreme situations, data races may simply end up silently corrupting user data. For all these reasons, data race detection remains one of the central concerns in parallel programming, in particular for shared memory programming models.

In previous papers [2,7], we presented the tool Archer [1], based on ThreadSanitizer (TSan) [8,9], which is able to find data races in OpenMP applications, that are run with the LLVM/OpenMP runtime on x86 machines. The fact which makes this tool unique from other approaches of available data race detection tools for OpenMP applications is that we cover almost all host-side OpenMP directives as provided in the OpenMP 4.5 specification. To make the tool portable

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 249–265, 2017.
DOI: 10.1007/978-3-319-65578-9_17

across OpenMP runtime implementations and hardware platforms, we want to base the annotation of OpenMP synchronization on OMPT events.

In this paper we investigate whether the information provided by OMPT is sufficient to derive all OpenMP synchronization semantics. We will describe OMPT based annotations of OpenMP synchronization. The annotations are provided as happened-before arcs, which can be understood by ThreadSanitizer, but also by the Valgrind based data race detection tool Helgrind. This approach is portable across OpenMP runtime implementations, as long as these implement and provide the necessary OMPT callback function invocations.

In Sect. 2 we look at OpenMP directives with synchronization semantics from a happened-before point of view. In Sect. 3 we describe the OMPT events, that we use to annotate the synchronization and how we specify the happened-before arcs. In Sect. 4 we discuss challenges we encountered on the way, implementing the tool and discuss information missing in the OpenMP tools interface.

2 Synchronization in OpenMP

According to the OpenMP specification [3]: "... if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit [...], then a data race occurs. If a data race occurs then the result of the program is unspecified."

To enable a data race detection tool to identify a data race, complete understanding of synchronization is needed. In this section we provide a summary of the synchronization concepts in OpenMP, as they need to be understood by an analysis tool, to identify synchronized memory accesses. In this paper we focus on data races that happen between threads on a host device. Thus, we do not consider constructs for offloading to an accelerator device.

2.1 The `parallel` Construct

When a thread encounters a parallel construct, the thread creates a team of threads to execute the parallel region. Each thread of the team executes the structured block of the parallel region within an implicit task.

Encountering the parallel construct *happens before* the execution of all implicit tasks of the team.

There is an implicit barrier at the end of the parallel region, which *happens before* the master thread continues execution.

2.2 The `barrier` Construct

The barrier in OpenMP applies for the innermost parallel team. On encountering a barrier construct, a thread cannot continue executing the implicit task until all threads in the team reached the barrier.

For all threads in the team, encountering the barrier construct *happens before* they continue execution of the implicit task.

2.3 The `reduction` Clause

The reduction clause provides a mechanism to reduce results at the end of a work-sharing region into a single value. The clause takes a reduction identifier to specify the reduction operation, the synchronization of the reduction is provided by the OpenMP implementation.

If no **`nowait`** clause is used on the same construct, the reduction *happens before* the end of the region. Otherwise the reduction *happens before* the next barrier.

2.4 The `critical` Construct

The critical construct provides mutual exclusion for the critical region. The critical construct can have a name, that provides mutual exclusion only for critical regions with the same name. The critical region is equivalent to getting a lock at the begin of the region and releasing the lock at the end, with different locks for different names and an extra lock for all unnamed critical regions. Thus, the synchronization semantics are the same as for Locking routines.

2.5 Locking Routines

OpenMP provides routines to init, destroy, acquire and release locks and nested locks. Locks provide mutual exclusion for code between acquiring and releasing a lock.

As a strict measure, a lock-set algorithm can be used to express the synchronization of critical region and locking routines. But lock-set is in general too strict and can lead to false positives The reason is that an application might implement *happens before* semantics in the locked sections. The alternative is to express locks with a happens before relation: Releasing a lock *happens before* acquiring the same lock.

This might over-estimate the synchronization semantics of the application and lead to omission of actual data races. This is a point, where large numbers of repetition and concurrency helps to stochastically execute the right interleaving of locked regions, so that the race can still be observed.

2.6 The `ordered` Construct

The ordered construct provides mutual exclusion for the ordered region. Additionally, the ordered construct also provides an ordering for the execution.

Thus, when observing the execution of an OpenMP program, the end of an ordered region *happens before* the begin of the next iteration of the same ordered region.

2.7 The `task` Construct

When a thread encounters a task construct, the thread generates a task from the associated structured block. The thread might execute the thread immediately, or defer the task for later execution.

Encountering the task construct *happens before* the execution of the task. The end of a task region *happens before* the next barrier of the team finished synchronization. Without further clauses or constructs, there is no more synchronization at the end of a task.

2.8 The `taskwait` Construct

The taskwait construct lets the encountering task wait for completion of all direct child tasks that this task created before encountering the taskwait.

Finishing all child tasks *happens before* the taskwait regions ends and the task can continue execution.

2.9 The `taskgroup` Construct

The taskgroup construct lets the encountering task wait at the end of the task group region for completion of all child tasks this task created in the taskgroup region and their descendants.

Finishing all child and descendant tasks *happens before* the taskgroup regions ends and the task can continue execution.

2.10 The `depend` Clause

The depend clause provides synchronization for task as the provided *in*, *out*, and *inout* dependencies define constraints for the scheduling of tasks. A depend clause can have a list of storage locations, which describe *in* or *out* dependencies. The end of a task with an *in* dependency on a storage location **x** *happens before* the start of any task with an *out* or *inout* dependency on the same storage location **x**. The end of a task with an *out* or *inout* dependency on a storage location **x** *happens before* the start of any task with an *in*, *out*, or *inout* dependency on the same storage location **x**.

To summarize, only *in* dependencies with the same storage location **x** do not synchronize. All other dependencies with the same storage location **x** synchronize.

2.11 Untied Tasks

Deferring a task *happens before* scheduling the same task again. This is especially important for untied tasks, that can migrate from one thread to another thread after being deferred during execution.

2.12 The `flush` Construct

The flush construct makes a thread's temporal view of memory consistent with memory and enforces a specific ordering of memory operations. The flush construct takes an optional list of variables, the *flush-set*. With the right combination of loads, stores and flushes, an application programmer can achieve fine-grain synchronization. Modeling the semantics of flushes with plain happened-before relation introduces synchronization which possibly hides any data race. A better approach for handling flushes is discussed by Lidbury and Donaldson [5]. They extend ThreadSanitizer to understand and handle C++11 flush semantics.

3 OMPT Events for Synchronization

In this section we explain the synchronization events provided by the OpenMP tools interface as it is integrated into the preview of the OpenMP specification 5.0 [4]. Since we implemented our prototype along with the LLVM/OpenMP runtime implementation, we used the version of OMPT, that is implemented there. The latest specification of OMPT describes events as points of interest in the execution of a thread. Tool callback functions are implemented in a tool and invoked by the runtime when a matching event happens. Multiple events might trigger the same callback; the tool can differ the events by some *kind* and *endpoint* arguments provided with the callback invocation. On tool initialization the OpenMP runtime implementation provides information to the tool, whether requested callback invocations are provided or not. For some groups of events invocation is mandatory, for some it is optional.

3.1 Team Related OMPT Events

The following events mark the synchronization points for a team from the creation of the team to the end:

- *parallel-begin*
- *implicit-task-begin*
- *barrier-begin*
- *barrier-end*
- *implicit-task-end*
- *parallel-end*

On a *parallel-begin* event, we generate a new team information object and start a happened-before arc for the team.

On an *implicit-task-begin* event, we generate a new task information object and end the happened-before arc for the team. This synchronizes the team creation.

On a *barrier-begin* event, we start a happened-before arc on an address from the team's information object. This event is specified to happen before the actual synchronization of the barrier.

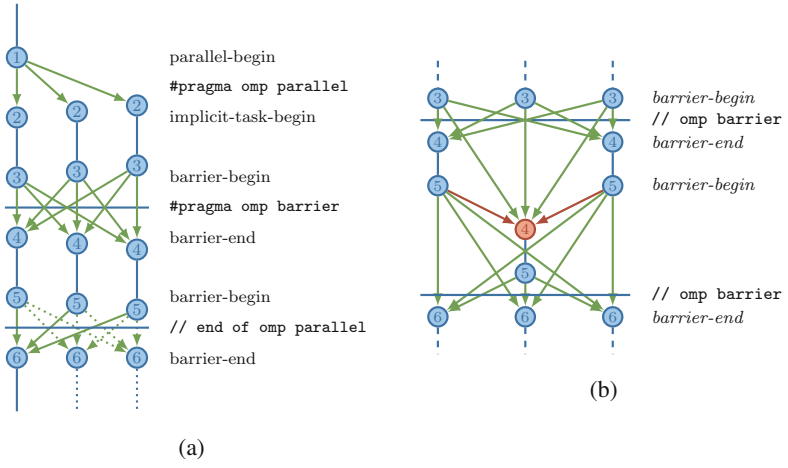

Fig. 1. (a) Happened-before arcs in a parallel region with explicit barrier and implied barrier at the end. (b) If a thread returns late from the barrier code (red *barrier-end* (4)), others might be already in the next barrier. In this case, we would add wrong happened-before arcs, if all barriers use the same token for the happened-before annotation (Color figure online)

On the *barrier-end* event, we end the happened-before arc on the same address from the team's information object. Since there is no synchronization between the barrier end event and the next barrier begin event, it is possible as depicted in Fig. 1b, that a thread of the team reaches the next barrier before another thread finished the previous barrier. Therefore, consecutive barriers should use distinct synchronization tokens. The OpenMP specification states that all threads in a team need to participate on each barrier, so we use two addresses for barriers in the team information object and each implicit task toggles between the two addresses.

The parallel region ends with an *implicit-task-end* event and a *parallel-end* event where we free the task and team information objects. The synchronization at the end of the region happens solely in the implied barrier at the end of the region. This is the second barrier in Fig. 1a.

As a missing piece in OMPT we will discuss the OpenMP reduction clause in Sect. 4.

3.2 Task Related OMPT Events

The following events mark the synchronization points for a task from the creation of a task to the end:

- *task-create*
- *task-dependences*
- *task-schedule*

– *taskwait-end*
– *taskgroup-begin*
– *taskgroup-end*

On a *task-create* event, we generate a new task information object and start a happened-before arc for the generated task. This synchronizes the task creation with the execution of the task. If this event is invoked before all data are copied to the task data structures, there might be some false data race alerts. Especially the copying of first-private data, which is then accessed by the task, might be a problem. See Fig. 2 for an illustration of the task-related events and happened-before synchronization.

On a *task-dependences* event we save all dependences information into the task information object for later use.

On the first *task-schedule* event for a new task, we end the happened-before arc from the generation of the task. Further, we iterate over all task dependences and end happened-before arcs for all dependences. If the dependency is an *in* dependency, we only end happened-before arcs from *out* or *inout* dependencies on this storage location. If the dependency is an *out* or *inout* dependency, we

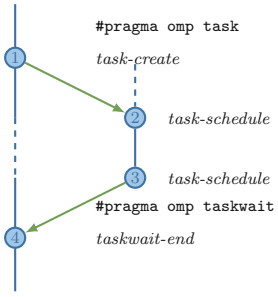

Fig. 2. Execution of a task happens after the task was generated from the parent; in case the parent task does a taskwait, the taskwait finishes after the generated task finished; end of taskgroup is similar

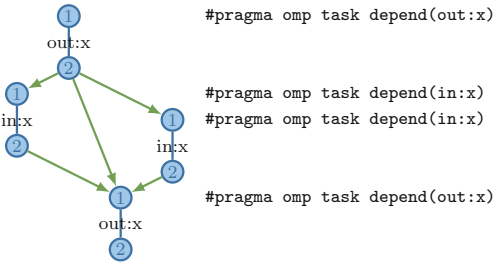

Fig. 3. This is the dependency graph for a set of tasks which were created with *out*, *in*, *in*, and *out* dependency on x; the end of a task with *out* dependency happens before all *task-begin* of tasks with a dependency on the same address. Tasks with the same *in* dependency run concurrently.

end happened-before arcs from all dependencies on this storage location. See Fig. 3 for an illustration of the dependencies-related events and happened-before synchronization. This also highlights the necessity to store the dependency information from task creation until task completion.

If the **prior_task_status** signals completion of the previous task, we start happened-before arcs for the completed task:

- towards a potential **taskwait** of the parent task
- if the task is in a **taskgroup** towards the end of the taskgroup
- if the task has dependencies, an arc per dependency.

On a *taskwait-end* event, we end the happened-before arc from all child tasks. We use a common token for all child tasks, so this is a single operation.

On a *taskgroup-begin* event, we push a taskgroup information object on the taskgroup stack of the encountering task. The stack is necessary because multiple taskgroup regions might be closely nested within a task. All child tasks inherit the taskgroup stack on task generation, so they know about their enclosing taskgroup.

On a *taskgroup-end* event, we end the happened-before arcs of all child tasks, targeting to the taskgroup end. Then we pop the taskgroup from the stack of taskgroups.

3.3 Locking Related OMPT Events

The following events mark the begin and end of mutual exclusion:

- *acquired-lock*
- *acquired-nest-lock-first*
- *acquired-critical*
- *acquired-atomic*
- *acquired-ordered*
- *released-lock*
- *released-nest-lock-last*
- *released-critical*
- *released-atomic*
- *released-ordered*

The latest OMPT specification consolidates all above events into a single callback for acquired and released with a **kind** argument for the kind of synchronization. For the happened-before synchronization, we only use the **wait-id** argument, so the handling of events is symmetric for all kind of mutex events.

On an *acquired* event, we end a happened-before arc, that starts on a previous *released* event.

To represent the synchronization semantics of locks in a data race analysis, it is important to start and end the happened-before arc inside of the locked region. Otherwise, another thread might already enter a locked region, before the released information is available. To reduce the potential overhead of an OMPT tool, the *released* event is invoked after the lock was released and there is no *releasing* event in OMPT. We discuss in Sect. 4.1 how we worked around this issue.

3.4 OMPT Flush Event

The flush event doesn't fit into the semantics of the previously discussed event groups. As touched in Sect. 2.12, happened-before semantics are too strict. But omitting the handling of flush, we experience false reports on data races in applications that use flush for synchronization. Implementing the right semantics for flush in our tool is subject of future work. But for now, we found that the information provided by the flush event is not sufficient for data race analysis as we will discuss in Sect. 4.6.

3.5 Team and Task Information Structures

We create an information object for each team and each task, which we store in the runtime scope of this team or task using the **`parallel_data`** and **`task_data`** fields provided by OMPT. In this section we detail on the necessary members of these objects. Both kinds of objects contain tokens, that we use to annotate different synchronization points.

A team object contains:

- two tokens for **`barriers`**, the tasks of the team use them alternating; we also use one of the tokens for the fork of the team.

A task object contains:

- a token for the **`task`**, that is used for the annotation, task-create before task-execution and task-deferring before rescheduling,
- a token for **`taskwait`**, which is used to annotate synchronization between the end of all child tasks and the taskwait,
- a **`barrier index`**, that toggles between odd and even barrier count,
- a **`reference count`** for direct child tasks, the object is only freed when the task and all child tasks finished execution,
- a reference to the **`parent`** task object,
- a reference to the **`implicit`** task object in the stack next to this task,
- a reference to the currently active **`taskgroup`** object,
- a copy of the *list of dependencies* and a *dependency count*.

A taskgroup object contains:

- a token for the *taskgroup*,
- a reference to the enclosing taskgroup.

4 Implementation Challenges and OMPT Shortcomings

In this section we discuss challenges, potential pitfalls and open issues which we encountered implementing the synchronization annotations in an OMPT-based tool.

4.1 Annotation of Locking

For TSan a happened-before annotation consists of writing memory at the start of the happened-before arc and reading the memory at the end of the arc. If the memory access is not synchronized, expressing the happened-before arc fails, since the read possibly happens before the write. For the annotation of locking this means, that the annotation needs to take place, while the thread owns a lock, that prevents the other thread from entering the locked region.

OMPT only provides the events *acquiring* (i.e. asking for the lock), *acquired* (when the lock is acquired) and *released* (after the lock was released) of a lock. OMPT does not provide a *releasing* event to safe the potential overhead in the critical path of execution. As depicted in Fig. 4a we would need to describe a happened-before arc from a *releasing* event to the next *acquired* event. And an arc from a *released* event to the *acquired* event goes potentially backwards in time.

As work-around for this issue we set an own mutex in each *acquired* event, before we end the happened-before arc and release the mutex in the matching *released* event after we started the happened-before arc. This approach is depicted in Fig. 4. This way we can guarantee that we annotate the end of a happened-before arc only after we annotated the begin of the happened-before arc. Since the OpenMP runtime already acquired a lock, we don't expect lock contention. It just might be the case, that the previous locked region still holds the mutex to finish the *released* event.

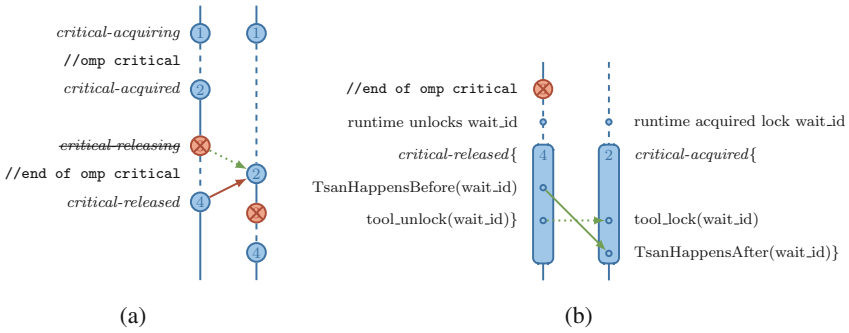

Fig. 4. (a) OMPT doesn't provide a *releasing* event. Using the *released* event to start the happened-before arc potentially results in a happened-before arc backwards in time. (b) We use an additional lock in the tool, to extend the exclusive region into the *released* callback. This way we can express the proper happened-before semantics.

4.2 Annotation of Task Dependencies

As discussed in Sect. 3.2, the synchronization behavior is different for *in* and *out* dependencies. The end of a task with an *in* dependency happens before a task begins with the same *out* dependency. The end of a task with an *out* dependency

happens before a task begins with the same *in* or *out* dependency. That means, at the task begin with an *in* dependency, we need to differ the happened-before arcs that come from *in* or *out* dependencies.

So, we need two different tokens for starting the happened-before arc of *in* dependencies and *out* dependencies. This token need to be common knowledge of all task using the dependency and for TSan the requirement for a token is that it needs to be a valid memory address of the process. For this reason, it is natural to use the address of the dependency as the token to annotate the happened-before arc. Since we need two tokens, we use the address provided as dependency and the address next to this address, assuming that applications will not use byte-sized variables as dependencies.

4.3 Ordered Construct with Depend Clause

For the online analysis that we apply in our data race detection tool, we rely on the scheduling decision provided by the runtime. We simply annotate any acquire of an ordered construct to happen after any release of the same ordered construct. This might be an overestimation and potentially hide data races. Since the depend clause allows the runtime to schedule multiple ordered regions at the same time, our tool might detect races in these concurrently executed regions. A tool which performs post-mortem analysis might not be able to observe this runtime decision and would assume mutual exclusive execution of all ordered regions in a loop. To improve precision of the analysis, we suggest to extend the notion of OMPT dependences to cover also the ordered construct.

4.4 Taskwait Construct with Depend Clause

Similar as with the ordered construct, we currently overestimate the synchronization effect of a taskwait construct with depend clause. In the analysis we assume that all tasks that finished before the taskwait region ends are synchronized by this taskwait region. With the additional information about the depend clause, the analysis would be more precise.

4.5 OMPT Events of Reductions

The current specification of OMPT provides no events for a reduction. The OpenMP specification does not require a specific point in the application execution, where the reduction needs to take place. Also an OpenMP implementation has a lot of freedom to implement the reduction algorithm, which results in various scenarios of memory access patterns. Threads might accumulate the own value to another thread's reduction value, threads might fetch other thread's reduction value and accumulate at the own reduction value, a master thread might collect all reduction values. The reduction might also be implemented solely with atomic operations.

We propose the following events for the implementation of reductions:

- *release-reduction:* thread will not touch reduction variable after this event
- *reduction-begin:* begin of reduction operations
- *reduction-end:* end of reduction operations

We think, that *release-reduction* and *reduction-end* can share the same callback function. The callback function needs to provide information about the local copy of the reduction variable.

The LLVM/OpenMP runtime implements most reductions inside the synchronization of the barrier. So as a temporary workaround, we ignore memory accesses inside of OpenMP barriers. If a task is scheduled in the barrier, we turn of ignoring memory accesses and turn it back on, when the barrier gets active again. This works in most cases for this specific runtime, but we don't expect this to be a portable workaround.

4.6 Information on Flush-Set

The current specification of the flush event as of TR4 only provides information on the source code of the flush (*codeptr_ra*) and the current thread, but no information on the provided *list* argument, which describes the flush-set of the flush operation. To derive the right flush semantics for data race detection, this information would be necessary.

We propose to extend the definition of **ompt_callback_flush_t** by an array of pointers, an array of length and a size argument:

```
1 typedef void (*ompt_callback_flush_t) (
2   ompt_data_t * thread_data,
3   const void * list_item,
4   size_t * list_item_length,
5   int list_length,
6   const void * codeptr_ra);
```

5 Implementation Results

To evaluate the overhead introduced by the TLC-aware data race analysis, we run SPEC OMP 2012 [6,11] on a machine with Intel Xeon E5-2650 v4 CPUs with 12 cores. We bind all threads to the same socket using **OMP_BIND=close** and **OMP_PLACES=cores**. Since the tool introduces a runtime overhead of about 2–20x – in some cases up to 125x – we only use the train dataset, which is the medium size for this SPEC benchmark.

ThreadSanitizer is optimized to run fast for race-free programs. If TSan detects data races, handling the report introduces significant runtime overhead. Printing the report happens under mutual exclusion to guarantee readable output without interleaving from multiple threads printing at the same time. Furthermore, TSan filters the output, so the report function also compares the latest finding with previous reports. Because of the filtering, TSan typically prints

reports only in the first few iterations; later races would mainly be duplicates. For actual debugging a user would typically interrupt the execution after some reports were printed, fix the issue and restart execution.

For better comparison we measure the overhead for the plain analysis without generating reports. Also ThreadSanitizer suggests this mode for benchmarking. In this mode, TSan intercepts all memory accesses, logs the memory access, analyses the memory access for potential data races. Also synchronization information is processed. The only difference from the normal mode is that in case of a detected data race TSan returns like there was no race instead of processing the report.

We use the LLVM/clang compiler 4.0 for the C/C++ codes and gfortran 6.2.0 for the Fortran codes. Both compilers provide the flag **`-fsanitize=thread`** to activate the compile time instrumentation for ThreadSanitizer. For the OpenMP runtime we use the LLVM/OpenMP runtime of the OpenMP tools subcommittee that implements the TR4 interface of OMPT.

5.1 Overhead Results

In Fig. 5 we plot the slowdown of the tool, which is runtime with tool divided by runtime without tool. We set the x-axis to 1, which is the normalized runtime of the application, i.e., the bar represents the tool overhead. As depicted, the overall measured slowdown is in the 2–20x range as claimed in the ThreadSanitizer documentation ("5–15x"[10]). But there are a some exceptions. Looking into the specific applications, this increased overhead mainly comes from fine-grain synchronization. In Table 1 we list some statistics important for the analysis tool. The two benchmarks where the tool shows overall high runtime overhead are 359.botsspar and 370.mgrid331. Both applications run for less than a second.

In this short time 359.botsspar already creates a large number of tasks. The synchronization for tasks happens from task to task. Hence, most of the time only one OS thread is involved. Another reason for a higher overhead is the use of untied tasks in this application. Since the tasks have no further task scheduling point, the tasks can only execute straight to the end. The code that the compiler

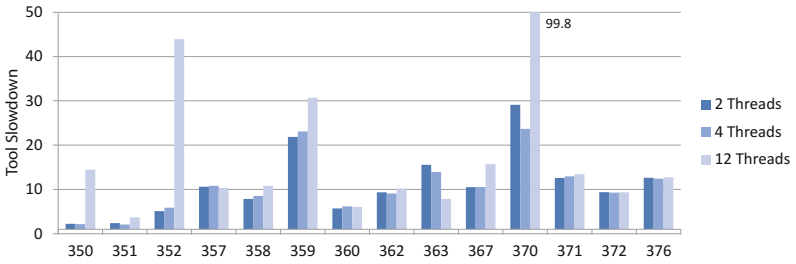

Fig. 5. Runtime overhead for executing SPEC OMP 2012 with ThreadSanitizer and synchronization annotations based on OMPT events

Table 1. OMPT synchronization events during the execution of SPEC OMP 2012, size train on 12 threads

application	runtime (s)	lang	parallel	barrier	task			other
					implicit	create	switch	
350.md	181	F	47	564	564	1		
351.bwaves	34.8	F	610	7320	7320	1		atomic: 384
352.nab	23.26	C	12428	350376	105048	2		
357.bt331	27.13	F	1011	21912	12132	1		
358.botsalgn	1.47	C	1	24	12	4951	9900	
359.botsspar	.14	C	1	12	12	88351	353400	taskwait: 200
360.ilbdc	72.93	F	1004	12048	12048	1		
362.fma3d	11.3	F	4367	52416	52404	1		atomic: 17292
363.swim	4.84	F	5601	86412	67212	1		atomic: 9600
367.imagick	11.17	C	13	138	138	1		
370.mgrid331	.35	F	6383	104784	76596	1		atomic: 48 ordered: 240
371.applu331	3.85	F	517	21432	6204	1		
372.smithwa	3.18	C	4	60	48	1		flush: 32369
376.kdtree	15.6	C++	4	72	48	1.53×10^9	3.06×10^9	taskwait: 1.53×10^9

generates for the untied task leads to a total of 4 task switches per task. This creates double the synchronization cost as for tied tasks.

370.mgrid331 creates more than 6000 parallel regions in just 0.3 s. Each parallel region ends with an implicit barrier, according to the data, about every third parallel region contains an additional barrier. For the happened-before analysis a barrier means a store to the same synchronization clock from every participating thread and a load afterwards. The writes to the synchronization clock need to be locked, so the synchronization cost for the barrier grows linearly with the number of threads, additionally we can expect increasing lock contention for a bigger number of threads. With less threads, chances are higher that a thread already finished the store when another thread arrives at the barrier. This results in the big increase of overhead for 12 threads.

For 352.nab we see another spike for 12 threads. This application also has a lot of barriers, which lead to the same issue as discussed for 370.mgrid331. For both applications the strong scaling contributes to the issue; with increasing number of threads, the work per thread decreases. This means the frequency of barriers also increases with number of threads. These two linear effects multiply and lead to quadratic overhead.

376.kdtree is the only application in the benchmark that uses OpenMP tasks in a recursive algorithm. This results in 1.5 billion of task in the train size. In an average execution with 12 threads, this application has a maximum number of about 550 concurrent tasks, counting tasks that are created, but not finished. For recursive algorithms with OpenMP tasks, at some point task creation gets too expensive compared to the workload; at this point, applications can use a serial cut-off. The remaining recursion is executed in a serialized fashion. 376.kdtree implements the cut-off by using **`#pragma omp task if`** with a dynamic condition. This means the task cannot be deferred and executes immediately. Taking this information into account, there are only 1.5 million tasks that are not unde-

ferred. By handling the undeferred tasks in a special way, we were able to reduce the runtime for 376.kdtree with 12 threads and ThreadSanitizer from about 450 s to 200 s. This reduced the overhead from 30x to 13x.

Finally, there is another spike for 350.md. This application is compute-bound on this machine since the problem size fits into cache and hence even 12 threads are not sufficient to exhaust the memory bandwidth. For smaller number of threads, this leads to the low runtime overhead with the tool. The code balance changes with the additional memory accesses coming from ThreadSanitizer, which adds about 4 times the memory foot print.

5.2 Data Race Results

Running the analysis, we were able to detect a data race in 367.imagick, which is caused by a concurrent write to a shared variable inside of a parallel region in `magick_decorate.c:492`. Making this variable private for the parallel region would resolve the data race.

Further, we could detect data races in 371.applu331. For this application we had the problem, that it uses custom synchronization on the base of conditional variables and flushes (in `syncs.f90`). The tool reports data races for the accesses of the conditional variable. By annotating these parts of the code, we were able to feed ThreadSanitizer with the synchronization information. With this annotations in the code, ThreadSanitizer only reports actual data races:

- `blts.f90:76` read after write in `blts.f90:66`, caused by the `do nowait` and the access to `v(1, i, j-1, k)`.
- `buts.f90:77` read after write in `buts.f90:243-247`, caused by the access to `v(1, i, j+1, k)`, `v(2, i, j+1, k)`, ...

372.smithwa is the other application that uses flushes to implement synchronization with conditional variables. For this application we don't see reported data races after the annotation of the synchronization in the application. We reported the identified data races to the SPEC group.

For some of the Fortran applications we see warnings about lock-order inversion coming from libgfortran. Because the file accesses in the application only happen in the serial part, the lock-order inversion is a benign issue. It is a known issue with ThreadSanitizer, that it reports lock-order inversion, although only a single thread accesses the lock.

6 Conclusions

In this paper we discussed whether OMPT provides sufficient information to derive all synchronization semantics needed for data race detection. We based the analysis on a happened-before based model. But we think, the observations would also apply for a different analysis model, based on lock-set or plain analysis of OpenMP flush semantics. We implemented a data race detection tool based on OMPT. With OMPT based annotations, the tool passes most of the tests in

our test suite. We pointed out three missing pieces of information in the OMPT interface, that is information about reduction, information about depend clause on taskwait and ordered constructs, and information on flush-set for flushes. We provided guidance on how to apply on-the-fly analysis for OpenMP mutual exclusion with the missing *releasing* event.

Further, we discussed the necessary OMPT events, to derive the synchronization information for data race analysis. To enable data race analysis based on these events, an OpenMP implementation needs to implement and provide callback invocation for these events. The issue here is that some of these callback invocations are optional according to current specification. This affects especially the events for taskwait, taskgroup, barrier and locks. If a data race detection tool cannot rely on these events, the advantage of portability across OpenMP implementations is gone. Therefore we suggest to make these callback invocations mandatory in the OpenMP specification.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the paper.

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-PROC-730143). Part of this work was possible under funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

1. Archer project and source code. https://github.com/PRUNERS/archer
2. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: ARCHER: effectively spotting data races in large openmp applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, 23–27 May 2016, pp. 53–62 (2016)
3. OpenMP Architecture Review Board: OpenMP Application Program Interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
4. OpenMP Architecture Review Board: TR4: OpenMP Version 5.0 Preview 1. http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf
5. Lidbury, C., Donaldson, A.F.: Dynamic race detection for C++11. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 443–457 (2017)
6. Müller, M.S., et al.: SPEC OMP2012 — an application benchmark suite for parallel systems using openMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 223–236. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30961-8_17
7. Protze, J., Atzeni, S., Ahn, D.H., Schulz, M., Gopalakrishnan, G., Müller, M.S., Laguna, I., Rakamaric, Z., Lee, G.L.: Towards providing low-overhead data race detection for large openMP applications. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, 17 November 2014, pp. 40–47 (2014)

8. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA 2009, pp. 62–71. ACM, New York (2009)
9. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 110–114. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29860-8_9
10. The Clang Team: Clang 5 documentation: Threadsanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html
11. Brian Whitney: SPEC OMP2012 documentation. https://www.spec.org/omp2012/Docs/