Thread-Local Concurrency: A Technique to Handle Data Race Detection at Programming Model Abstraction[*]

Joachim Protze
RWTH Aachen University, ITC
Aachen, Germany
protze@itc.rwth-aachen.de

Martin Schulz
TU Munich
Munich, Germany
schulzm@in.tum.de

Dong H. Ahn
LLNL
Livermore, CA
ahn1@llnl.gov

Matthias S. Müller
RWTH Aachen University, ITC
Aachen, Germany
mueller@itc.rwth-aachen.de

ABSTRACT

With greater adoption of various high-level parallel programming models to harness on-node parallelism, accurate data race detection has become more crucial than ever. However, existing tools have great difficulty spotting data races through these high-level models, as they primarily target low-level concurrent execution models (e.g., concurrency expressed at the level of POSIX threads). In this paper, we propose a novel technique to accurately detect those data races that can occur at higher levels of concurrent execution. The core idea of our technique is to introduce the general concept of Thread-Local Concurrency (TLC) as a new way to translate the concurrency expressed by a high-level programming paradigm into the low execution level understood by the existing tools. Specifically, we extend the definition of vector clocks to allow the existing state-of-the-art race detectors to recognize those races that occur at the higher level of concurrency with minor modifications to these tools. Our evaluation with our prototype implemented within ThreadSanitizer shows that TLC can allow the existing tool to detect these races accurately with only small additional analysis overheads.

ACM Reference Format:
Joachim Protze, Martin Schulz, Dong H. Ahn, and Matthias S. Müller. 2018. Thread-Local Concurrency: A Technique to Handle Data Race Detection at Programming Model Abstraction. In *HPDC '18: International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3208040.3208056

[*]This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-744962).

1 MOTIVATION

Effectively and efficiently exploiting on-node parallelism is crucial to delivering requisite computational power to today's applications. As this need has arisen from a wide range of domains, not only from High Performance Computing (HPC), but also from other areas of computing such as data-intensive and even technical computing, many high-level parallel programming models, in response, have emerged and have been adopted.

One of the main characteristics of these models is that they decouple user-level abstractions used to express concurrency from how the concurrent work units are run in parallel. A user can express the concurrency more productively using their richer and higher-level concurrency idioms, and the runtime system of these models then maps this to the lower levels of concurrent execution (e.g., POSIX thread level). In the last few decades, such decoupling has become so popular that these features are found in many models: e.g., non-blocking communication in MPI or PGAS approaches, tasks in OpenMP or Cilk, and offloading of execution to a co-processor with CUDA, OpenACC or OpenMP.

A data race condition is inarguably the most malignant form of parallel interaction in any shared memory programming model. This condition occurs when two or more threads can access shared data without proper synchronization, and at least one them modifies the data. The result can change depending on which thread wins this race. But they are notoriously difficult to catch, in part because they can be difficult to spot and reproduce through traditional testing. As the high level programming models exploit concurrency and parallelism, they are also subject to the adverse effects of the races.

Automatic data-race detection in general is one of the most widely studied problems in concurrent program design. And there are largely three common approaches. The most common data race detection techniques are happened-before analysis (e.g., ThreadSanitizer [22]), lock set analysis (e.g., Eraser [20]), or a hybrid approach that combines these two analyses (e.g., Helgrind [9]). While lock set analysis can potentially detect more data races, it is also prone to create more false alarms. Thus, in practice, happened-before analysis-based tools have gained most popularity.

Independent of the analysis approach, dynamic runtime data race detection tools must collect information about memory accesses and synchronization. Some tools like Helgrind and Intel Inspector use binary instrumentation at runtime to collect the information. ThreadSanitizer makes use of compile time instrumentation. This allows a targeted analysis of selected compilation units and ignore other parts of the code. When analyzing application code, the application developer needs to understand the issues in their own code and to a certain degree should assume that the runtime libraries he/she use behaves correct; debugging runtime libraries is not the application developer's business.

In the Archer project [4] we already started to introduce this abstraction for OpenMP applications. Synchronization information is tracked at the abstraction of OpenMP synchronization semantics [18] using the new OpenMP tools interface (OMPT) which is added in OpenMP 5.0. Archer registers callback functions for all OMPT synchronization events. In these callback functions we call into the ThreadSanitizer runtime to annotate the OpenMP synchronization semantics. The OpenMP runtime library is not further instrumented. This allows to exchange the OpenMP runtime implementation and still have the synchronization information, as long as the runtime provides the new OMPT interface. Unfortunately, the current implementation of ThreadSanitizer does not allow to completely ignore non-instrumented code. Function calls to some basic memory management functions, like memset or alloc, are always intercepted and can lead to false alarms in non-instrumented code.

For MPI this approach of ignoring the runtime library and annotating MPI semantics can avoid the issues of false alerts known from using Valgrind and Helgrind with OpenMPI [1]. We will discuss the synchronization semantics of MPI function calls in the next section. The memory access semantics of MPI calls can be reported to the analysis with the help of a PMPI wrapper.

Many tools implement happened-before analysis based on vector-clocks to distinguish synchronized and concurrent events. Events before the latest recorded synchronization are called synchronized, while events after the synchronization are considered concurrent. Using this distinction, the tool can then detect multiple concurrent accesses from different execution units (e.g., threads, processes, etc.) that happen to access the same memory location; with at least one of the concurrent accesses being a write access, the tool reports a race.

For this approach, a tool must maintain a vector clock for each concurrent execution unit and each vector clock needs an entry per concurrent execution unit. Current state-of-the-art tools focus on race detection for threaded codes and target a low-level thread model that typically matches operating-system-level entities. This means that the number of execution units, and with that the size of the vector clocks, equals the number of OS threads. As this number in most cases matches the number of hardware cores or threads, the vector clocks are guaranteed to be of reasonable size.

However, this approach forces users to reason about races that are meaningful at the OS level, rather than at the level of the programming abstraction they are using for concurrency: i.e., it ignores implicit synchronization given in programming models and reports races at an abstraction level that does not match the user's source code. This is another reason, why such tools generate false positives which programmers then manually have to sort through.

To overcome this, we need a new generation of race detection tools that recognize and operate at the same abstraction level the user uses to express his or her concurrency. This, however, turns any concurrent activity in the programming abstraction into a concurrent execution unit for the race detector. As we will show, even with medium sized workloads, this can lead to numbers of concurrency units that can be orders of magnitude higher than the concurrency at the operating system (OS) level, rendering the use of traditional vector clocks infeasible. Although the programming abstraction introduces new levels of concurrency, technically each hardware unit only executes one abstract execution unit at a time.

To overcome this challenges and provide feasible, low-overhead tools that can detect races at the abstraction level of the programming model, we introduce the general concept of *Thread-Local Concurrency (TLC)* as a new concept to enhance the operating system level of abstraction. As its main advantage, it provides a model to analyze concurrency within a single OS thread, allowing us to work at the user's level of abstraction with the cost of an OS level abstraction. In particular, this paper makes the following contributions:

- Introduce the concept of *Thread-Local Concurrency (TLC)* to reason about concurrent execution units within individual OS threads;
- Illustrate how we map the concurrency expressed in different programming paradigms to TLC;
- Explain how TLC analysis reduces memory complexity for vector clocks from $O(n^2)$ to $O(n * t)$ and the synchronization cost from $O(n)$ to $O(t)$. Where n is the concurrency provided by the programming paradigm and t is the visible concurrency at the operating system level.

We implement our new technique in ThreadSanitizer, a widely used, state-of-the-art race detector. We provide an extensive overhead evaluation and demonstrate how we can use our tool to detect data races that occur in OpenMP task parallel programs.

2 EXAMPLES OF CONCURRENCY ABSTRACTIONS

Modern programming models provide several mechanisms to express concurrency. Such capabilities are crucial to exploit the growing node concurrency we see in today's systems as well as to efficiently overlap computation and communication. However, such abstractions for concurrency also open the door for data races caused by the concurrency introduced by them.

In this section we discuss several use cases with different concurrency abstractions, in particular the concurrency semantics of MPI non-blocking communication and the concurrency introduced by the OpenMP task and target constructs. We show how data races affect these semantics and why conventional detection tools fail to adequately detect those. Each of the selected examples is a representative for a group of similar programming models to which the observations can be equally applied to.

2.1 Non-blocking Communication in MPI

In MPI, a non-blocking initiation function (like `MPI_Irecv` or `MPI_Isend`) is used to receive or send data in the background, while continuing the execution and potentially finishing some calculation instead of blocking until the communication is finished. A call to a

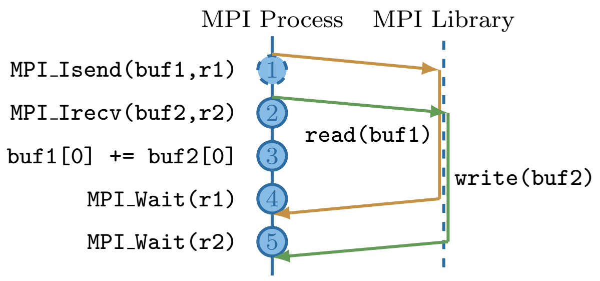


Figure 1: For non-blocking communication, the MPI library can read or write the buffer anytime between the initiation and completion call. Changing a send buffer or changing or reading a receive buffer in this interval leads to undefined behavior.

```
MPI_Isend(buf1, 1, MPI_INT, ..., &r1);
MPI_Irecv(buf2, 1, MPI_INT, ..., &r2);
buf1[0] += buf2[0];
MPI_Wait(&r1, ...);
MPI_Wait(&r2, ...);
```

Listing 1: Source code sketch for Figure 1

completion function (like `MPI_Wait`) synchronizes and finishes the communication.

For non-blocking communication with send semantics, the application is not allowed to change the memory in the send buffer after the initiation and before the completion call – as depicted in (1) and (3) in Figure 1. For non-blocking communication with receive semantics, the application is not allowed to access the memory passed to the MPI library as the receive buffer after the initiation and before the completion call – as depicted in (2) and (4) in Figure 1. Therefore, the read of `buf2[0]` and the write to `buf1[0]` each are in conflict with the related MPI communication call.

At the MPI abstraction layer, the non-blocking communication is an additional execution unit that forks with the initiation call, performs read or write operations representing the send or receive semantics, and joins with the completion call.

MPI implementations are free to use a communication thread which helps to make progress in communication while no MPI function is active. Depending on the implementation in the MPI library, data race detection tools working on the operating system abstraction level cannot detect a violating memory access as a data race if the MPI library executes communication on the application thread. Even for the case of an MPI implementation that executes communication on a helper thread, a tool might miss the access if the memory accesses inside the MPI library are not instrumented or the conflict is between multiple communication calls.

2.2 OpenMP Tasks

OpenMP Version 3.0 introduced tasks as a new concept. With the task construct, the application developer can explicitly split work into chunks, that either execute immediately or are deferred for later execution. Deferred tasks can be executed by any thread within a group or team of threads. Without further synchronization, any pair of tasks could be executed concurrently on different threads in the team.

Aside from explicit synchronization given by task dependencies or a `taskwait` construct, all tasks in a team should be seen as concurrent execution units. Consequently, accesses to the same memory in concurrent tasks qualify as data races if at least one of the accesses is a write operation. According to the memory model of OpenMP, the result of the program is undefined if there is any data race.

In Figure 2 we show the scheduling and happened-before arcs for various concrete executions of the code in Listing 2. In Figure 2a only the happened-before arcs (1,3), (2,5), (4,7) and (6,7) are given by the execution model of OpenMP. The write to *a* in the master region is synchronized with the write to *a* in the task, because the task is created after the first write to *a*. The two tasks, however, are not synchronized, so that the write to *b* in the two tasks is a data race. Similarly, the write to *c* is not synchronized, because the execution of the second task is not synchronized with the execution after the task creation. Adding dependency in/out clauses on *b* to the tasks and a taskwait construct after the second clause would resolve the data races and introduce exactly the happened-before arcs as in Figure 2b. The Figures 2b,c show potential schedules in case the parallel team consists only of one thread. In Figure 2b, the tasks are executed immediately after creation. In Figure 2c, the tasks are executed in the implicit barrier at the end of the parallel region. In both cases, the execution of the tasks is ordered due to the runtime scheduling decision. A data race detection tool that works at the operating system level cannot detect the data races on *b* and *c*, if the tasks are scheduled to the same OS thread.

2.3 OpenMP Target Offloading

OpenMP Version 4.0 introduced target offloading to a device as a new concept. With the target construct, a program can offload data and code to a target device. While not further specified in the OpenMP specification, a target device is in practice a co-processor or graphics card (GPU).

As Munchhalfen et al. discuss [14], various kinds of data races between accesses on host and target device are possible. If a target region changes memory and the host device reads the same memory without synchronization (target update), the result is unspecified. For an OpenMP target map clause, it is not specified whether OpenMP copies or directly maps the memory to the target device. For systems with shared memory between host CPU and target device, the latter is possible. Consequently, depending on the actual data mapping, the host device would see the old or the modified data. Even if the execution of the target code seems synchronized with host execution, the mapped data region should be seen as a concurrent execution unit. However, such races caused by concurrent accesses from CPU and GPU are not covered at all by current tools, which either entirely focus on CPU or GPU races individually.

2.4 Hybrid MPI+OpenMP

To utilize the compute capabilities of todays supercomputers, more and more applications augment their code by hybrid distributed and

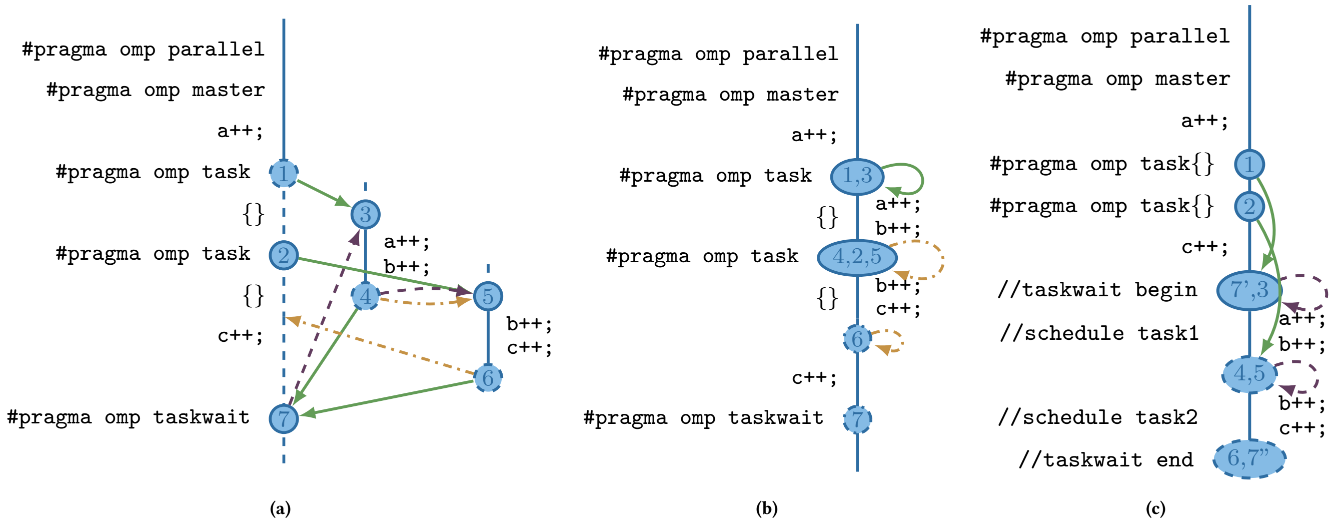


Figure 2: Synchronization information for OpenMP tasks. We assume shared(a,b,c). (a) Displays the synchronization information without concrete scheduling of tasks. Only the arcs (1,3), (2,5), (4,7) and (6,7) are provided by the OpenMP semantics. The additional arcs (4,5) and (6,.) are introduced by the immediate schedule in (b). The additional arcs (7,3) and (4,5) are introduced by the schedule at the taskwait in (c).

```
#pragma omp parallel
 {
  #pragma omp master
   {
    a++;
    #pragma omp task shared(a,b)
     { a++; b++; }
    #pragma omp task shared(b,c)
     { b++; c++; }
    c++;
    #pragma omp taskwait
   }
  sleep(2); // emulate load
 }
```

Listing 2: Source code for Figure 2

shared memory parallelization. Often this is a combination of MPI and OpenMP, but also MPI and Pthreads can be found. While MPI is still dominant in HPC, also several PGAS approaches draw attention which can also be combined with multi-threading paradigms.

Hybrid parallel programs can be challenging for data race detection tools. To understand all possible data races of a hybrid application, the tool needs to understand both paradigms. Analyzing the paradigms separately is not sufficient as we can show with the short code example in Listing 3.

If the code is executed by a single OpenMP thread, the code is executed sequentially on all MPI processes. The array is initialized, then swapped with the communication partner in the `MPI_Sendrecv` and finally reduced into a single value. An MPI-only tool cannot find a data race here, since there is no race.

If the code is executed by a single MPI process, i.e., ignoring the MPI parallelism, the MPI operation is an in-place memcopy and the MPI implementation might not touch any memory. In the case that the MPI implementation actually copies the memory, a data race detection tool might detect a race between the initialization of the

```
#pragma omp parallel
{
  #pragma omp for nowait
    for(j = 0; j < PSIZE; j++)
      array[j] = rank * j + 2;

  #pragma omp master
    MPI_Sendrecv(array, PSIZE, MPI_INT,
         size-rank-1, 42, array, PSIZE, MPI_INT,
         size-rank-1, 42, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);

  #pragma omp for reduction(+:sum)
    for(j = 0; j < PSIZE; j++)
      sum += array[j];

  #pragma omp master
    printf("Rank %i total %i\n", rank, sum);
}
```

Listing 3: Hybrid OpenMP+MPI datarace

array and the copy. Although the copying in `MPI_Sendrecv` writes back the value that was read before, another thread initializing an element of the array might write to the element in between, so that at the end the uninitialized value is written back to the array.

If an OpenMP data race detection tool completely ignores the memory accesses by MPI, it cannot find a data race in this example.

This example emphasizes that a data race detection tool needs to understand all parallel programming paradigms used in the application and also understand their interactions to identify the data races resulting from hybrid parallelization.

3 THREAD-LOCAL CONCURRENCY (TLC)

Existing data race detection tools fail for the use cases in the previous section. These tools work only at the abstraction level of the operating system and hence cannot properly function under advanced concurrency scenarios. Instead we need tools that operate at the level of the abstraction used to specify and implement the concurrency. However, for this to work, any concurrent element expressed in the programming abstraction (e.g., each OpenMP task, each MPI asynchronous operation, or each OpenMP target construct) has to be represented using its own concurrent execution unit.

For a traditional tool based on vector clock analysis, this means that the size of the vector clock, which grows with the number of execution units, becomes unmanageable. Further, each execution unit also needs to maintain its own vector clock to store and express synchronization with the other execution units, which will also be unmanageable. For 376.kdtree, one of the example applications in our measurements, we have seen as many as 550 concurrent tasks during the execution. This has a significant impact on both space and compute requirements: the memory complexity of this approach is $O(n^2)$ with n being the number of concurrent execution units; while the complexity of a synchronization operation grows linearly with the size of the vector clock, i.e., $O(n)$. Both are not scalable. For these scalability reasons, there is currently no tool available that implements this fine-grain tracking of concurrency.

To overcome this problem and to provide a scalable solution, we introduce the new concept of *Thread-Local Concurrency (TLC)*. The observation of execution shows, that each individual OS thread can only execute one thing at the same time. The individual concurrent execution strings from the previous section can be seen as time slicing parallelization within an OS thread. Further, a concurrent execution string is initiated or interrupted at some point and starts or continues execution at some later point. Anything that happens between initiation or interruption and starting or continuation is concurrent. We can interpret an interruption as initiation of the continuation and the end of the current execution; this means we only need the concept of initiating and starting of TLC execution.

In Figure 2c the first task is initiated in (1), the second task is initiated in (2). The first task then starts execution, which should be seen as concurrent to (1,7'), when finished the second task starts execution which should be seen as concurrent to (2,4).

With TLC we express the concurrency of the slices to understand the additional concurrency at the right level of abstraction while mapping the analysis to the same number of execution units (and with that vector clock length) as OS based approaches. This way we can analyze the additional concurrency without explicitly introducing additional execution units, allowing us to combine the benefits of both solutions.

We define *Thread-Local Concurrency* as all phases (time slices) in the preceding execution of a thread that need to be seen as concurrent to the current execution. This means for events X and Y that are observed on the same thread with X observed earlier than Y, still $X \nrightarrow Y$, if X is in a thread-local concurrency phase of Y.

4 HAPPENED-BEFORE WITH TLC

Lamport [11] defined the *happened before* relation as an irreflexive partial order of events:

Definition: The relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by an other process then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \nrightarrow b$ and $b \nrightarrow a$.

We kept the notion of processes in the definition, although we mean any execution unit and in our use cases especially OS-threads. We extend this happened before relation by the concept of TLC:

First we define a *TLC-phase*: All events between initiation of TLC execution i and the starting of this TLC execution s on the same thread create a new TLC-phase for s. The first event on a process has no TLC-phase.

Definition: The relation $\rightarrow_{TLC}$ on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, a comes before b, **and** $\boldsymbol{a}$ **is not in an TLC-phase of** $\boldsymbol{b}$, then $a \rightarrow_{TLC} b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by **any** process then $a \rightarrow_{TLC} b$. **(3) With** $\boldsymbol{i}$ **as the initiation of TLC execution and** $\boldsymbol{s}$ **as the starting of this TLC execution in the same thread holds** $\boldsymbol{i \rightarrow_{TLC} s}$**.** (4) If $a \rightarrow_{TLC} b$ and $b \rightarrow_{TLC} c$ then $a \rightarrow_{TLC} c$. Two distinct events a and b are said to be *TLC-concurrent* if $a \nrightarrow_{TLC} b$ and $b \nrightarrow_{TLC} a$.

An important thing to note is, that according to this definition, TLC-phases propagate from initiation to starting TLC execution and also across processes. On the other hand, one transitive path from a to b is sufficient, so that rule (2) leads to cancellation of TLC-phases and only an intersection of all these TLC-phases is relevant. Furthermore, rule (3) in combination with the definition of the TLC-phase mean, that no rule (2) synchronization between i and s is relevant in s and onwards.

To achieve the pursued limited memory and analysis cost, we restrict the analysis in our implementation to consider only the latest local TLC-phase on each thread, we call this the *TLC-window*. For the implementation this means, that in addition to the traditional synchronization information we only need the information about the TLC-window interval (i, s).

In this section we briefly give an overview of general vector clock based happened-before analysis. Then we discuss a step-by-step work flow of the analysis with a TLC-aware vector clock. Finally, we start with a transition system which describes the implementation of a distributed vector clock and extend this transition system to introduce the concept of a TLC-aware vector clock.

4.1 Happened Before with Vector Clocks

A concrete implementation of a happened-before analysis based on vector clocks (VC) in ThreadSanitizer is described by Serebryany et al. [21]. This tool works at the operating system abstraction level and therefore assumes threads as the smallest building block for concurrency. This results in the assumption that an event *happened before* another event on the same thread if it is observed before the other event. The analysis in this tool is based on distributed vector clocks. Each thread stores its own state of synchronization in a VC, so there is one *VC per thread*. The i-th value in this VC expresses the logical time of the latest synchronization with the thread i. Events on thread i with a smaller logical time happened before the synchronization, events with a larger logical time are concurrent with the current execution on the local thread. Synchronization between execution units is achieved by updating the vector clock at a *waiting* execution unit with the values from the *signaling* execution unit. To implement the transitivity of happened-before, this update is a pairwise max operation on all VC entries. In the implementation, the signaling thread stores the own VC into a *synchronization clock (SC)* and the waiting thread updates the own VC from the SC when the wait succeeded. So there is a *synchronization clock* bound to each signal. In practice, there is for example a SC bound to each lock, the SC is identified by some common knowledge, like the address of the lock. Unlocking happens before the next successful lock, so these operations store and load from this SC. In [18] we describe in detain, how we bind SC to various synchronization constructs for OpenMP.

A general problem with plain happened-before analysis is the risk to miss actual races. Without further possibilities to express synchronization, an unlock-lock sequence must be interpreted as a signal-wait sequence. This means that any activity before the lock on the first thread appears to be synchronized with any activity after the lock on the other thread.

By limiting our analysis to the local TLC-window, we do not loose accuracy compared to plain happened-before analysis, because for signaling between threads, the happened-before relation is equivalent with the common one. But we gain the ability to express and analyze concurrency on the thread. The limitation to the local TLC-window therefore compares to precision loss when going from lock-set analysis to happened-before analysis.

4.2 Using a TLC Vector Clock

Figure 3 illustrates the use of a TLC vector clock to analyze the memory accesses in Listing 2. For a simplified diagram, only a single thread is noted in the vector clock. The clock value is followed by the entries for the begin and the end of the TLC window. With each operation the clock value is incremented. The master thread increments a, which is logged as a write access at clock=3. Then the thread creates two tasks, which are deferred for later execution. The tool stores the vector clock and binds the information to the newly created tasks. Finally, the master thread increments c and the tools logs the write at clock=6 before the thread needs to wait for the created tasks to be finished. The taskwait construct is a task scheduling point, so one of the tasks gets scheduled. Semantically, everything between creating the task and scheduling the task is concurrent with the execution of the task. Therefore, the TLC window for task $t1$ starts with the creation time (4) and ends with the scheduling time (7). The latest access to a happened before the task creation, which can also be derived from comparing the access time (3) with the vector clock: $3 < 4 \Rightarrow$. For this task, the tool just logs the access times for a and b. The TLC window for task $t2$ starts with the creation time (5) and ends with the scheduling time (10). The latest access to b was during the execution of t1; comparison of the access time with the TLC window shows $9 \in (5, 10) \Rightarrow$ unsynchronized. Similarly was the latest access to c after the task creation. The comparison of the access time with the TLC window shows $6 \in (5, 10) \Rightarrow$ unsynchronized. Since in both cases, a write access was involved, the tool would report a data race.

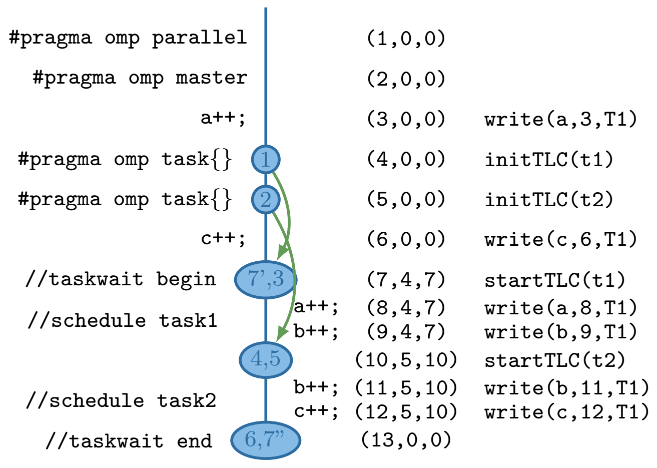
Handling the taskwait-end is not TLC specific. To provide the tool with all necessary synchronization information, all tasks store their vector clock into a taskwait specific synchronization clock when they finished execution. This clock is loaded at the taskwait-end and results in the (13,0,0) vector clock.

4.3 Vector Clock Defined as Transition System

The transition system $\mathcal{T}$ describes a vector clock system as it is used for happens before analysis in ThreadSanitizer, with a clock vector for each of n threads and additional m synchronization clock (SC) vectors. The synchronization clock vectors are used to store (send/signal) and load (receive/wait) clock vectors.

$$\mathcal{T} = (States, \rightarrow_{\mathrm{VC}}, L_0)$$

$States = \mathbb{N}^{n\times(n+m)}$ constitutes the state space of $n + m$-tuples $(t_0, \ldots, t_{n-1}, s_0, \ldots, s_{m-1})$ where $t_i, s_j \in \mathbb{N}^n$ represent the vector clock for each thread respectively synchronization point. We address the kth entry of the vector clock t_i by $t_{i,k}$. We use the initial state L_0 with $t_{i,k} = 0, s_{j,k} = 0$ for all $i = 0..(n-1), j = 0..(m-1), k =$



```
#pragma omp parallel          (1,0,0)
  #pragma omp master          (2,0,0)
              a++;            (3,0,0)    write(a,3,T1)
 #pragma omp task{}           (4,0,0)    initTLC(t1)
 #pragma omp task{}           (5,0,0)    initTLC(t2)
              c++;            (6,0,0)    write(c,6,T1)
  //taskwait begin            (7,4,7)    startTLC(t1)
                       a++;   (8,4,7)    write(a,8,T1)
  //schedule task1     b++;   (9,4,7)    write(b,9,T1)
                             (10,5,10)   startTLC(t2)
                       b++;  (11,5,10)   write(b,11,T1)
  //schedule task2     c++;  (12,5,10)   write(c,12,T1)
   //taskwait end             (13,0,0)
```

Figure 3: TLC vector clock illustration. The figure displays the vector (vector clock, concurrency window). The analysis for the second increment of a shows that the previous access at 3 was before t_B, i.e., is synchronized. The analysis for the second increment of b shows that the previous access at $\mathbf{9} \in (\mathbf{5}, \mathbf{10})$, i.e., was in the TLC window and therefore is a data race. Similar for c: $\mathbf{6} \in (\mathbf{5}, \mathbf{10})$.

0..$(n-1)$. We define the transition relation $\rightarrow_{\text{VC}} \subseteq \textit{States} \times \textit{States}$ as the smallest binary relation on *States* satisfying the rules:

(1) Clock tick on thread i:

$$\overline{(\ldots,(\ldots,t_{i,i},\ldots),\ldots) \xrightarrow{\text{tick}}_{\text{VC}} (\ldots,(\ldots,t_{i,i}+1,\ldots),\ldots)}$$

(2) Signal on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{signal}}_{\text{VC}} (\ldots,t_i,\ldots,Up(s_j,t_i),\ldots)}$$

(3) Wait on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{wait}}_{\text{VC}} (\ldots,Up(t_i,s_j),\ldots,s_j,\ldots)}$$

The update function Up on clock vectors as used in rules (2) and (3) is defined as:

$$Up(a,b) = (max(a_1,b_1),\ldots,max(a_n,b_n))$$

4.4 TLC Vector Clock

We now extend the concept of the vector clock system by a concurrency interval for a thread. For this, we extend the clock vector by two values to mark the begin and end of the interval and one value to identify the local thread:

$$C = (\mathbb{S}, \rightarrow_{\text{TLC}}, L_0)$$

The set of states $\mathbb{S} \subseteq \mathbb{N}^{(n+3)\times(n+m)}$ constitutes the state space of $n+m$-tuples $(t_0,\ldots,t_{n-1},s_0,\ldots,s_{m-1})$ where $t_i, s_j \in \mathbb{V} \subseteq \mathbb{N}^{n+3}$ represent a state vector—consisting of a vector clock, concurrency interval, and concurrency thread-id—for each thread (t_i) respectively synchronization point (s_j). We address the kth entry of the vector clock of t_i with t_{i,C_k}, we address the whole vector clock of t_i with $t_{i,C}$, further we address the concurrency window with $t_{i,W} = [t_{i,B} : t_{i,E}]$, and the thread-id for the window with $t_{i,T}$. In summary the state vector is denoted as $t_i = (t_{i,C}, t_{i,W}, t_{i,T})^T$. The initial state L_0 of the system is $t_{i,C_k} = 0$, $s_{j,C_k} = 0$, $t_{i,W} = [0 : 0]$, $s_{j,W} = [0 : 0]$, $t_{i,T} = i$, $s_{j,T} = \infty$ for all $i = 0..(n-1), j = 0..(m-1), k = 0..(n-1)$.

We define the transition relation $\rightarrow_{\text{TLC}} \subseteq \mathbb{S} \times \mathbb{S}$ as the smallest binary relation on States $\mathbb{S}$ satisfying above rules (1)-(3) as well as the following additional rules:

(4) TLC-Signal on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{tlc-signal}}_{\text{TLC}} (\ldots,t_i,\ldots,Up_s(s_j,t_i),\ldots)}$$

(5) TLC-Wait on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{tlc-wait}}_{\text{TLC}} (\ldots,Up_t(t_i,s_j),\ldots,s_j,\ldots)}$$

(6) TLC-Init on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{tlc-init}}_{\text{TLC}} (\ldots,t_i,\ldots,t_i,\ldots)}$$

(7) TLC-Start on thread i, using synchronization clock j:

$$\overline{(\ldots,t_i,\ldots,s_j,\ldots) \xrightarrow{\text{tlc-start}}_{\text{TLC}} (\ldots,Start(t_i,s_j),\ldots,s_j,\ldots)}$$

The rules (4) and (5) use the update function $Up_{t/s}$ defined as:

$$Up_{t/s} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}, \text{ with}$$

$$Up_t(a,b) \stackrel{def}{=} Norm_t(Up'(a,b))$$

$$Up_s(a,b) \stackrel{def}{=} Norm_s(Up'(a,b))$$

The normalization function $Norm_{t/s}$ fixes meaningless window intervals that end before they begin and is defined as:

$$Norm_{t/s} : \mathbb{V} \rightarrow \mathbb{V}, \text{ with}$$

$$Norm_t(a) \stackrel{def}{=} \begin{cases} (a_C, [0:0], a_T)^T & : \text{if } a_B >= a_E \\ a & : \text{else} \end{cases}$$

$$Norm_s(a) \stackrel{def}{=} \begin{cases} (a_C, [0:0], \infty)^T & : \text{if } a_B >= a_E \\ a & : \text{else} \end{cases}$$

The actual update function Up' is defined as:

$$Up' : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}, \text{ with}$$

$$Up'(a,b) \stackrel{def}{=} \begin{cases} \begin{pmatrix} max(a_C,b_C) \\ [b_B : b_E] \\ b_T \end{pmatrix} & : \text{if } a_T >= n \\ \begin{pmatrix} max(a_C,b_C) \\ [max(a_B,b_B) : \\ min(a_E,b_E)] \\ a_T \end{pmatrix} & : \begin{matrix} \text{if } a_T = b_T \wedge \\ a_E < b_{a_T} \end{matrix} \\ \begin{pmatrix} max(a_C,b_C) \\ [max(a_B,b_{C_{a_T}}) : a_E] \\ a_T \end{pmatrix} & : \begin{matrix} \text{if } (a_T = b_T \wedge \\ a_E >= b_{a_T}) \\ \vee (a_T \neq b_T \wedge \\ a_T < n) \end{matrix} \end{cases}$$

Rule (7) uses the initializing function $Start$ defined as:

$$Start : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}, \text{ with}$$

$$Start(a,b) \stackrel{def}{=} \begin{pmatrix} r_C \begin{cases} r_{C_j} = b_{C_j} : & \text{if } j \neq b_T \\ r_{C_j} = a_{C_j} : & \text{if } j = b_T \end{cases} \\ [b_{C_{b_T}} : a_{C_{b_T}}] \\ b_T \end{pmatrix}$$

The definition of the Up function for (2) and (3) is extended to set t_B and t_E to 0, t_T is set to tid, for synchronization clocks s_T is set to ∞. With this modified Up function, the transitions (2) and (3) keep the semantic for happened-before arcs without TLC-window.

The Up_t and the Up_s function in the transitions (4) and (5) are used to create a happened-before arc (rule 2) from one TLC-aware state to another TLC-aware state. As described in the introduction of this section, the resulting TLC-phases are an intersection of the current state and the incoming message, considering also the time since the last synchronization as a TLC-phase.

Since we only analyze the local TLC-window, the resulting TLC-window is an intersection of the TLC-windows in the two state vectors (second case of Up'), again considering also the concurrency phase after the last synchronization (third case of Up'). If the two state vectors of a transition belong to different threads, we consider only the concurrency phase after the last synchronization to cut with the TLC-window (third case of Up'). The first case covers the situation, where no TLC-window is stored in a SC.

The transition (6) is used to recycle a synchronization clock that was potentially used for another synchronization before and store the state of initiation of a TLC execution. With this optimization, the number of synchronization clocks can be cut down to the maximum number of concurrent execution units at any point during the execution.

The transition (7) starts execution with thread-local concurrency information. The previously active synchronization information is dropped and the state of the synchronization clock is loaded into the thread state. Using the clock values from the synchronization clock means that upcoming analysis will work on the base of the old synchronization information. For the thread with TLC window, the current clock value is used in the new vector clock, the TLC window is built using the synchronization clock value and the current clock value.

5 IMPLEMENTATION OF TLC

ThreadSanitizer (TSan), as described in section 4.1, is implemented in latest GNU and LLVM/clang compilers and uses happened-before analysis based on distributed vector clocks. Each thread has a local vector clock (VC) that stores the latest synchronization with each of the other threads. For each memory access, a thread writes a log entry that associates the local entry of the VC and the own thread id with this memory access. Further, the thread checks for unsynchronized, colliding memory accesses on the same memory location. For this, the thread iterates over the other log entries on this memory location. In case of a potential collision, the thread compares the clock value in the log with the clock value in the local VC for the thread id in the log. The comparison provides the information whether the other access happened before the current access, or concurrently with the current access.

We base the implementation of our new TLC approach on the ThreadSanitizer extension Archer that already annotates OpenMP synchronization for ThreadSanitizer and is described by Atzeni et al. [4], the annotations are described in detail in [18]. The extension consists of implementation of TLC-analysis in the ThreadSanitizer runtime and changing the OpenMP synchronization annotation in Archer to include TLC-init and start for tasks. In this section we describe the interface to annotate application synchronization and how we integrate our new technique into ThreadSanitizer and the OpenMP runtime annotations.

5.1 Annotating Happened-Before

Newer versions of LLVM expose a simple interface to annotate synchronization or memory accesses in libraries or applications for ThreadSanitizer. This interface is defined in the header file **`llvm/Support/Compiler.h`**. The interface consists of the functions:

- **`TsanHappensBefore(cv)`**
- **`TsanHappensAfter(cv)`**
- **`TsanIgnoreWritesBegin()`**
- **`TsanIgnoreWritesEnd()`**

The latter two functions allow to explicitly suppress ThreadSanitizer analysis for parts of the code. The functions **`TsanHappensBefore`** and **`TsanHappensAfter`** allow to annotate the beginning and the end of an happened-before arc (rule 2) if both calls use the same pointer argument *cv*.

As depicted in Algorithm 1, the **`TsanHappensBefore`** function updates the VC that is associated with the synchronization point *cv* with the local VC. This implements the $\overset{\text{signal}}{\rightarrow}_{\text{VC}}$ transition as described in the previous section. In the same way, the

Algorithm 1 Effect of happened-before annotation in ThreadSanitizer

```
procedure TsanHappensBefore(cv)
    $cv.clock \leftarrow max(cv.clock, thread.clock)$
end procedure
procedure TsanHappensAfter(cv)
    $thread.clock \leftarrow max(cv.clock, thread.clock)$
end procedure
```

`TsanHappensAfter` function updates the local VC from the VC of the synchronization point. This implements the $\overset{\text{wait}}{\rightarrow}_{\text{VC}}$ transition.

5.2 Annotate Happened-Before with TLC

We extend the interface of ThreadSanitizer by the following functions, that represent the new relations (4)-(7) as introduced in the previous section:

- **`TsanHappensBeforeTLC(cv)`**
- **`TsanHappensAfterTLC(cv)`**
- **`TsanInitTLC(cv)`**
- **`TsanStartTLC(cv)`**

We further extend the runtime library of ThreadSanitizer. We add the TLC window to the thread state and to the synchronization clock. The core data race analysis kernel now checks whether a colliding memory access on the same thread was observed in the TLC window. In this case we also report a data race. And finally, we add an implementation for above new interface functions to the ThreadSanitizer runtime.

5.3 TLC-Aware Analysis for OpenMP

In previous work we described the annotations necessary to apply conventional happened-before analysis on OpenMP applications [18]; this work already handles the synchronization semantics of OpenMP task execution including task dependencies. For OpenMP we call the ThreadSanitizer annotation functions building on the new OpenMP Tools interface (OMPT), as described in the draft release for OpenMP 5.0 [16]. As a part of the OpenMP tools working group we upstreamed an implementation of OMPT into the LLVM/OpenMP runtime [2] available with the 6.0 release.

Here we describe the TLC specific changes to this previous work: On the task-create event, we store the state of the thread into a synchronization clock bound to the new task using the **`TsanInitTLC`** function. On the task-schedule event, which marks the start of execution of the task, we load the synchronization clock into the thread state using the **`TsanStartTLC`** function. With these functions, the vector clock in the thread state is set to the vector clock as seen during task creation.

The **`TsanInitTLC`** for task-create replaces the **`TsanHappens-Before`** used by Archer to annotate that task creation happened before task execution, the **`TsanStartTLC`** for task-schedule replaces the **`TsanHappensAfter`** in the original Archer annotations. We use these annotation calls as the fallback if the tool is used with a compiler version that does not provide the new ThreadSanitizer API.

To allow synchronization of sibling tasks, OpenMP offers the concept of task dependencies to describe DAGs. Archer already handles task dependencies by introducing a happened-before arc for each task dependency. For TLC-aware analysis, we replace the **TsanHappensBefore** by **TsanHappensBeforeTLC** and **TsanHappensAfter** by **TsanHappensAfterTLC**.

5.4 Annotation of New Memory

To fully map the memory access semantics from the programming paradigm abstraction to the tool, we also need to handle the concept of memory allocation and deallocation.

For the OpenMP task use case, TLC-enabled analysis would report false alerts on stack usage between consecutive task executions, since ThreadSanitizer cannot know that the stack of the previous task was cleared in between. The internal annotation interface of ThreadSanitizer already includes an interface function to describe new memory, but no implementation:

- **TsanNewMemory(addr,size)**

Our implementation of this function deletes all memory access log entries that belong to the provided memory range. With this annotation function, we wipe all accesses to the stack above the task scheduling when the runtime switches tasks.

6 MEASUREMENTS

To have the feature of TLC-aware data race analysis available, we would like to apply ans upstream the necessary extensions to ThreadSanitizer. This extension could introduce overhead for applications that don't need this feature. To evaluate the overhead introduced by the TLC-aware data race analysis, we run SPEC OMP 2012 [13, 24] 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 tools compared introduce a runtime overhead of about 2-20x – in some cases up to 80x – we only use the train dataset, which is the medium size for this SPEC benchmark. We presented some similar measurements for the basic data race analysis in [18]. For a complete overview on synchronization characteristics of the SPEC OMP 2012 benchmark kernels we refer to that paper. Differences in the reported base runtime for some of the applications come from a recent kernel update (meltdown mitigation) and the use of a newer compiler and OpenMP runtime version.

We run ThreadSanitizer in benchmarking mode. In this mode, ThreadSanitizer 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 ThreadSanitizer returns like there was no race instead of processing and printing the report.

We use LLVM/clang compiler built from Feb 18 2018 sources from git for the C/C++ codes and gfortran 6.2.0 for the Fortran codes. Using gfortran for the Fortran applications was necessary since by the time of the implementation of the tool no Fortran compiler frontend for LLVM was available. Both compilers provide the flag `-fsanitize=thread` to activate the compile time instrumentation for ThreadSanitizer. Fortunately, the ThreadSanitizer implementations for GNU/gfortran and LLVM/clang are compatible and can be exchanged.

We use a modified ThreadSanitizer runtime library based on the runtime coming from clang where we implemented our additional annotations and the TLC-window the code can be found in tlc branch of [3]. To apply our modified ThreadSanitizer runtime library also to the Fortran applications, we compile the applications with gfortran, but link the applications with clang:

```
FC=gfortran -fopenmp -fsanitize=thread -O3
FLD=clang -lomp -fsanitize=thread -lgfortran
```

6.1 Measurement Results

In Figure 4 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, so that the remaining bar represents the tool overhead. The figure shows pairwise the results for measurement without and with TLC-aware analysis. The left bar of each pair shows the overhead of currently available ThreadSanitizer analysis, the right bar shows the overhead of our updated implementation. As depicted, in most cases the measured slowdown is in the 2-20x range as claimed in the ThreadSanitizer documentation ("5-15x"[23]). But there are a some exceptions that we discussed in detail in [18], here we focus on the additional overhead of the TLC-aware analysis.

6.2 Overhead of TLC-Aware Analysis

As mentioned above, in Figure 4 we also plot the slowdown for TLC-aware data race detection with ThreadSanitizer. The two expected influence factors for runtime overhead introduced by TLC-aware analysis are the additional three values in the state vector to be copied on synchronization, which means additional cost mainly for synchronization; as well as the additional comparison with the TLC-window for colliding local memory accesses. For applications without explicit tasks, this is only one comparison, but it still means that an extra value, the end of the concurrency window, is loaded and compared. This value also occupies a cache line, which potentially would not be touched without TLC analysis. For applications with explicit tasks, the previous access is compared with begin and end of the concurrency window. This introduces additional cost for applications which repeatedly access the same memory from the same thread.

For most applications, we don't see a significant change in runtime overhead. These applications don't show frequent synchronization that could lead to additional overhead. For 367.imagick we also expect most memory to be read only once while running over the pixels of an image, so there shouldn't be a lot of colliding memory accesses.

The highest relative increase in tool slowdown we see is about 4% for 2 threads executing 370.mgrid331 and for all thread counts executing 351.bwaves. We see about 3% for 360.ildbc, 362.fma3d and 363.swim.

6.3 Validating the ThreadSanitizer Reports

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.

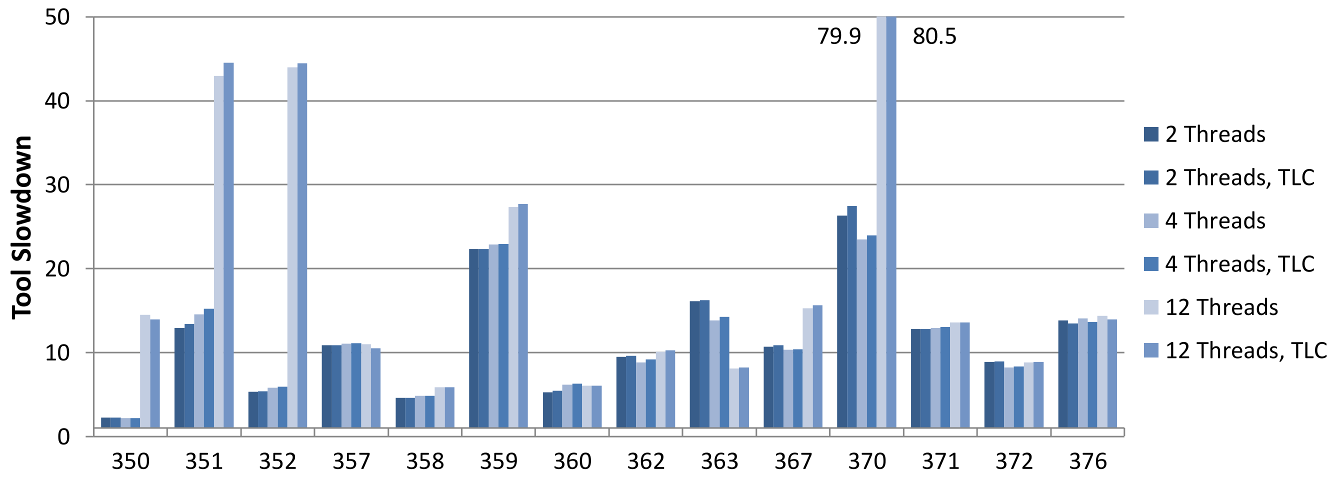


Figure 4: Runtime overhead for executing SPEC OMP 2012 with ThreadSanitizer and synchronization annotations based on OMPT events

```
36          neigh = iam - 1
37  !$omp flush(isync)
38          do while (isync(neigh) .eq. 0)
39  !$omp flush(isync)
40          end do
41          CALL AnnotateHappensAfter(__FILE__,
                __LINE__, isync(neigh))
42          CALL AnnotateHappensBefore(__FILE__,
                __LINE__, isync(neigh))
43          isync(neigh) = 0
44  !$omp flush(isync,v)
```

Listing 4: Annotation of flush-based customized synchronization in syncs.f90 of 371.applu331. Line numbers refer to the original source code.

371.applu331 and 372.smithwa are applications that use flushes to implement customized synchronization with conditional variables. By carefully annotating the synchronization semantics for these conditionals, we could avoid any false alerts for these applications; unfortunately, this kind of annotations needs to be done by the programmer, if the tool fails to understand handcrafted low-level synchronization.

An example for such an annotation is shown in Listing 4. Here we discuss the annotations for the `sync_left` function in syncs.f90 of 371.applu331. This file is also part of the LU application in the NAS Parallel Benchmarks [5]. The thread spins in the while loop on `isync(neigh)` until another thread changes the value to 1. With the annotation `AnnotateHappensAfter`, the tool is notified about the end of an happened-before arc. The annotation `AnnotateHappensBefore` starts another happened-before arc to the thread, that will wait for the variable to be set to 0. Having these annotations exactly at this place is crucial for correct analysis. The annotation in the `sync_right` function is analogical.

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.

Finally, in the initial reports for the two tasking applications we got reports on data races for access to stack memory. This issue was discussed in section 5.4, but shows that we can successfully detect data races between tasks which are scheduled on the same thread.

Algorithm 2 ThreadSanitizer annotations for non-blocking MPI communication

```
procedure MPI_ISEND(buf, req)
    PMPI_ISEND(buf, req)
    $req.buf \leftarrow buf$
    TSANSTORETLC(req.cv)
end procedure
procedure MPI_WAIT(req)
    PMPI_WAIT(req)
    TSANHAPPENSBEFORE(tmp)
    TSANLOADTLC(req.cv)
    TSANREADMEMORY(req.buf)
    TSANHAPPENSAFTER(tmp)
end procedure
```

7 APPLYING TLC TO OTHER USE CASES

In Section 2 we presented examples of concurrency abstractions introduced by parallel programming paradigms. In the previous sections we discussed how to apply TLC for OpenMP tasks and how this enables ThreadSanitizer to detect a new class of races. In this section we sketch how TLC enables ThreadSanitizer to apply analysis for the other two concurrency abstractions—non-blocking MPI communication and accelerator offloading.

7.1 Annotation of Non-Blocking MPI Communication

Non-blocking MPI communication, as sketched in Figure 1, is another use case that can be handled with a TLC window. The MPI interface provides an interception layer, called PMPI, that tools can use to wrap any MPI call with their own functionality. In Algorithm 2 we sketch how we use this to annotate the functions `MPI_Isend` and `MPI_Wait` to enable ThreadSanitizer to detect unsynchronized violating memory accesses of the application with the asynchronous access of the MPI library to the communication buffer.

For the initiation call, the information about the communication buffer and the current vector clock is stored and bound to the request handle.

For the completion call, the current vector clock is temporarily stored with the `TsanHappensBefore(tmp)` call. The call to `TsanLoadTLC(req.cv)` loads the synchronization information as present at the initiation time. The next call `TsanReadMemory(req.buf)` annotates the memory access semantics, this lets ThreadSanitizer log the memory access and analyze conflicting memory accesses for this memory range. Finally, the `TsanHappensAfter(tmp)` call restores the synchronization information from the begin of the completion call. We implemented Algorithm 2 as a prototype to detect data races in MPI applications. With this prototype, we could successfully detect a data race, when accessing the send buffer of an `MPI_Isend` before the synchronization with `MPI_Wait`. The prototype is very limited in function, since only isend and wait are considered, and only base types are supported. Therefore no evaluation with a real application is viable.

7.2 Annotation for OpenMP Offloading

The idea of annotating OpenMP offloading is quite similar to the annotations used for MPI in Algorithm 2. Mapping memory to the device is handled like the initiation for MPI non-blocking communication. When the memory is mapped back, it is annotated as a write access. If the target data region ends without mapping the memory back, the memory is annotated as read.

This strategy might overestimate the memory access pattern and possibly lead to false alerts. An example would be an algorithm that on the device writes to even entries of an array and reads the odd entries. At the same time the host can read the odd entries. The above strategy would report a race in this case.

If no device is available to offload a target region, the OpenMP runtime would execute the code of the target region on the host. In this case, we can annotate the execution of the target region to be concurrent with the code outside the target region while annotating synchronization as provided by OpenMP semantics.

8 LIMITATIONS OF THE TLC-WINDOW

In this section we discuss some limitations of the TLC window for data race analysis. The limitations range in both directions, omission of actual data races and false alerts for valid memory access patterns.

8.1 False Alerts

With the TLC approach, we relax the synchronization model coming from strict happened-before analysis and introduce some exceptions. The assumption for this approach is that there is no implicit synchronization between the execution of two applications. However, this assumption does not hold for all algorithms.

A valid OpenMP program can implement an algorithm, where each task works on a thread specific section of a shared array. In such a program, the allocation is synchronized with the accesses to the memory in the tasks, because the tasks are created after the allocation. Further, the execution can assume, that only one tasks is active on a thread at the same time. Therefore, the access to the array is also synchronized between the tasks. TLC-aware data race analysis would flag the accesses to the results array as races in the case that multiple tasks are scheduled to the same thread.

This issue is introduced by the assumption that tasks in a team execute concurrently when not explicitly synchronized. This means that also an analysis modeling each task as a concurrent execution unit, as briefly sketched in Section 3, would have the same issue, i.e., the issue is not caused by the simplification introduced by our TLC-aware analysis.

8.2 Omission of Data Races

To limit the memory cost of TLC-aware analysis, we use the TLC window, which restricts the analysis to a single concurrency phase. For the use cases discussed in Section 7 this does not imply any restriction. Multiple concurrency phases would occur when TLC-aware happened-before arcs are used. We carefully designed the transitions in Section 4 in a way that we restrict the stored concurrency phases to the latest phase, the TLC window. For OpenMP tasks, TLC-aware happened-before arcs are used to express the synchronization between parent and child task for task creation and to express synchronization by task dependencies. With the TLC window we can express the concurrency of the executing task E with previous tasks P_i that were accidentally scheduled onto the same thread. When the executing task creates a new task N, this new task has no synchronization with the tasks P_i, so N should be concurrent with P_i. When the task N is scheduled, the interval from creating the task to execute the task is the latest concurrency phase. With the single TLC window, we cannot express P_i to be concurrent with N.

Also in the case of immediate execution of the included tasks, like in Figure 2b, we cannot see the concurrency of the tasks, as the first task finishes execution before the second task is created.

To model TLC-aware analysis equivalent to an analysis that models tasks as independent execution units, we would need to transitively propagate concurrency information. Also, if N is scheduled on a different thread than E, we would need to keep the concurrency phase of executing P_i. Hence, for an equivalent model, we would need a list of concurrency intervals per thread, not only for the local thread.

9 RELATED WORK

Data race detection is a widely studied problem and it was shown to be NP-hard by Netzer and Miller [15]. Several tools exist to analyze data races in OpenMP applications, like Intel Inspector, Archer [4], Helgrind [9] and Oracle Thread Analyzer. Most of these tools do a good job for fork-join parallelism in applications that mainly use OpenMP parallel teams. Of these tools, only Archer understands the synchronization implied by OpenMP tasks and task dependencies. ThreadSanitizer as the underlying tool of this work, as well as Archer, implement the FastTrack algorithm [7] with a limited history of accesses per memory location, which makes it similar to iFT [8]. This not only makes memory overhead predictable in space and time, but also allows the use of all available processing units concurrently for execution.

Other work on data race detection for asynchronous execution proposes the SP-bag [6] and ESP-bag [19] algorithms. These approaches serialize the execution and implement a depth first search for the analysis of data races. This approach does not scale with the growing number of processing units available on today's and future machines. Further, applying the SP-bag algorithm to non-blocking MPI communication would mean serializing all MPI communication, which ultimately might introduce a deadlock in the MPI communication.

Some papers discuss how to exploit symmetries and regularities in OpenMP fork-join programs to improve scalability of data race detection [10, 12]. While this is not directly applicable to asynchronous tasks, especially the Offset-Span labeling suggested in [12] might be extended to asynchronous workloads. Although OS-labeling reduces the cost for exchange of synchronization information, it does not provide the constant memory complexity per memory access.

In recent years, for many applications in HPC memory bandwidth became the bottleneck. For this reason, it is hard to compare one approach with other approaches if there is no implementation available to obtain measurements with typical workloads on current machines.

An orthogonal approach to improve accuracy of happened-before based data race detection suggests active testing to enforce scheduling behavior that leads to data races [17].

10 CONCLUSIONS

In this paper we discussed data race detection for parallel programming paradigms at two abstraction levels. One level is the abstraction of the programming paradigm, the other level is the operating system and base language level. Many parallel programming paradigms introduce additional concurrency. We presented a new approach to map the concurrency provided by various programming paradigms down to the operating system layer. For this purpose, we introduced the new concept of *Thread-Local Concurrency* or *TLC*. We showed the applicability of the approach to implementations of OpenMP tasks and MPI non-blocking communication. Based on overhead measurements using the SPEC OMP 2012 suite, we were able to show that the introduced overhead of this approach is low enough to make adding this feature permanently to the ThreadSanitizer runtime is feasible. This will enable a tool chain that can analyze hybrid MPI+OpenMP applications for any kind of MPI or OpenMP specific data race at the same time.

The implementation of TLC-aware ThreadSanitizer and TLC-aware analysis for OpenMP are available in [3].

REFERENCES

[1] Documentation of OpenMPI on Valgrind usage. https://www.open-mpi.org/faq/?category=debugging#valgrind_clean.

[2] OpenMP: Support for the OpenMP language. http://openmp.llvm.org.

[3] TLC implementation. https://github.com/PRUNERS/compiler-rt (branch tlc) and https://github.com/PRUNERS/openmp (branch archer-tlc).

[4] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 53–62, 2016.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[6] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA*, pages 1–11, 1997.

[7] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 121–133, 2009.

[8] Ok-Kyoon Ha and Yong-Kee Jun. An efficient algorithm for on-the-fly data race detection using an epoch-based technique. *Scientific Programming*, 2015:205827:1–205827:14, 2015.

[9] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. Helgrind+: An efficient dynamic race detector. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–13, 2009.

[10] Young-Joo Kim, Sejun Song, and Yong-Kee Jun. VORD: A Versatile On-the-fly Race Detection Tool in OpenMP Programs. *International Journal of Parallel Programming*, 42(6):900–930, 2014.

[11] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. pages 558–565, July 1978.

[12] John Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.

[13] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, G. Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. SPEC OMP2012 - an application benchmark suite for parallel systems using openmp. In *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*, pages 223–236, 2012.

[14] Jan Felix Münchhalfen, Tobias Hilbrich, Joachim Protze, Christian Terboven, and Matthias S. Müller. Classification of Common Errors in OpenMP Applications. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, pages 58–72, 2014.

[15] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *LOPLAS*, pages 74–88, 1992.

[16] OpenMP Architecture Review Board. TR6: OpenMP Version 5.0 Preview 2. http://www.openmp.org/wp-content/uploads/openmp-tr6.pdf.

[17] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 51:1–51:12, New York, NY, USA, 2011. ACM.

[18] Joachim Protze, Jonas Hahnfeld, Dong H. Ahn, Martin Schulz, and Matthias S. Müller. OpenMP Tools Interface: Synchronization Information for Data Race Detection. In *Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings*, pages 249–265, 2017.

[19] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 368–383, 2010.

[20] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[21] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[22] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler. In *Runtime Verification*, pages 110–114. Springer, 2012.

[23] The Clang Team. Clang 5 documentation: ThreadSanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html.

[24] Brian Whitney. SPEC OMP2012 Documentation. https://www.spec.org/omp2012/Docs/.