OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs

Bradley Swain
Computer Science and Engineering
Texas A&M University
College Station, U.S.A
brad@tamu.edu

Yanze Li
Computer Science and Engineering
Texas A&M University
College Station, U.S.A
yanzeli@tamu.edu

Peiming Liu
Computer Science and Engineering
Texas A&M University
College Station, U.S.A
peiming@tamu.edu

Ignacio Laguna
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, USA
ilaguna@llnl.gov

Giorgis Georgakoudis
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, USA
georgakoudis1@llnl.gov

Jeff Huang
Computer Science and Engineering
Texas A&M University
College Station, U.S.A
jeff@tamu.edu

Abstract—We present OMPRACER, a static tool that uses flow-sensitive, interprocedural analysis to detect data races in OpenMP programs. OMPRACER is fast, scalable, has high code coverage, and supports the majority of commonly used OpenMP features by combining state-of-the-art pointer analysis, novel value-flow analysis, happens-before tracking, and generalized modelling of OpenMP APIs.

Unlike dynamic tools that currently dominate the field of data race detection, OMPRACER achieves almost 100% OpenMP code coverage using static analysis to detect a broader category of races without running the program or relying on specific input and runtime behaviour to trigger the race. Moreover, OM-PRACER has competitive precision with representative dynamic tools such as Archer and ROMP: it passes 105 out of the 116 cases in DataRaceBench with a total accuracy of 91%. OMPRACER has been used to analyze the Exascale Computing Project (ECP) proxy applications containing over 2 million lines of code in under 10 minutes and has revealed a previously-unknown race in them.

I. INTRODUCTION

OpenMP is a *de facto* standard for on-node parallelism in HPC, targeting both multicore CPUs and accelerators such as GPUs, and also serving as the backend of high-level programming models, such as RAJA [1] and Kokkos [2]. While writing parallel programs using OpenMP is easy, writing them correctly is as hard as any other multithreaded programming, which is notoriously prone to race conditions and other concurrency errors such as deadlocks and atomicity violations. Debugging race conditions in OpenMP is particularly challenging because the non-deterministic races can be perplexed by the OpenMP runtime, which is hidden from the static program.

The majority of recent work in OpenMP race detection has focused on dynamic tools. There are a number of dynamic tools in use today, including Archer [3], ROMP [4], SWORD [5], TSAN [6], Intel Inspector [7], and Helgrind [8]. The dynamic tools typically detect a data race by tracing memory accesses per thread, running a program with a specific configuration of number of threads and input, to report a data race if writing a memory location violates *happens-before* [9] ordering semantics. This approach has several drawbacks: 1) it is slow because of the high overhead associated with memory tracing; 2) detection depends on the specific thread and input configuration, so it may miss data races that do not manifest with a specific configuration—testing every possible thread setting and input is practically infeasible; 3) it can be sometimes difficult to accurately pinpoint the cause of a data race in the source code since this method involves redirection through debugging information for the executable. While recent work [4] improves the accuracy of runtime data race detection by analyzing memory accesses at the level of concurrent *logical tasks* of OpenMP, i.e., it detects races independent of the threading configuration, this detection method still depends on input and incurs the overhead of running the program traced.

Previous work has also proposed a small number of static verification tools for OpenMP, including DRACO [10], LLOV [11], and ompVerify [12]. These tools put heavy emphasis on proving that a section of code is race free. While powerful, such strong guarantees come at a cost: these tools are either restricted to a subset of OpenMP features, such as affine loops, are too expensive to scale to large programs, or must operate in a limited scope of the program.

In this paper, we present OMPRACER, a new data race detection tool that performs flow-sensitive, interprocedural static analysis to find races in OpenMP programs. OMPRACER combines several novel techniques designed for OpenMP, including a static happens-before graph built upon a generalized modeling of OpenMP APIs, an efficient race detection engine through hybrid happens-before and lockset analyses, an interprocedural value-flow analysis for array indices, together with a state-of-the-art pointer analysis. OMPRACER detects races independently of both the threading configuration and input, and it integrates seamlessly with the compiler toolchain without requiring any annotation or other intervention from the

user. Compared to existing static tools, OMPRACER does not aim to be a verification tool; instead it is explicitly designed to be a tool capable of supporting the majority of OpenMP features that can scale to real-world large complex applications with millions of lines of code.

The specific contributions of this work are:

- A new, state-of-the-art data race detection tool, named OMPRACER, that performs detection at compile time using static analysis. We design OMPRACER so that its accuracy is independent of the number of threads or input configuration. OMPRACER is deployed as an extension to the Clang/LLVM compiler and cmake toolchain, requires no user intervention other than compiling the program through it, and pinpoints exactly the source code line and source code variable for which the race occurs.
- A novel technique to detect OpenMP races that extracts OpenMP semantics to build the static happens-before graph at the level of *logical threads* and understands OpenMP-defined data sharing semantics. Combining OpenMP semantic analysis with a set of novel algorithms for reasoning about pointer aliases and value flows in an interprocedural setting enables OMPRACER's to achieve both scalability and precision.
- An extensive evaluation of OMPRACER using DataRaceBench [13] and HPC proxy applications comparing with other state-of-the-art tools, including the widely used tool Archer [3], the recently developed OpenMP-specific tool ROMP [4], as well as a more recent static OpenMP race detector LLOV [11]. Besides OMPRACER being superior than those tools by detecting races regardless of thread and input configuration and avoiding the overhead of traced execution, OMPRACER shows high accuracy and high precision on DataRaceBench benchmarks.

We have applied OMPRACER to the Exascale Computing Project (ECP) proxy applications containing over 2 million lines of code. OMPRACER is fully automated to run on all of them under 10 minutes. In our evaluation of the proxy application miniAMR [14], OMPRACER found a previously-undetected data race (we elaborate the race in the next section). We also applied our tool to a high profile covid simulation code, finding several new races that are missed by dynamic race detectors. This demonstrates the effectiveness of OMPRACER in being ready to be used in realistic OpenMP applications.

II. MOTIVATING EXAMPLES

In this section, we illustrate challenges in OpenMP race detection and highlight the advantages of our approach.

A. Challenges

Input-dependent Races. Listing 1 shows an example of a data race that depends on a specific branch being executed. An array A and its length N are given as inputs. There is a data race on A[0], but only when A is over a certain length. In order for a dynamic tool to detect the race, an array A given as input must be large enough that the branch inside of the parallel for loop is executed. If the wrong input is given and the branch is not taken, it is impossible for a dynamic tool to detect the race.

```
1  int *A; int N;
2  load_from_input(A, &N);
3  #pragma omp parallel for shared(A)
4  for(int i = 0; i < N; i++) {
5      A[i] = i;
6      if (N > 10000) { A[0] = 1; }
7  }
```
Listing 1. Input Dependent Race

Although the example in Listing 1 is simple, this problem in principle applies to large projects. Different paths within a program may access different locations in memory and dynamic tools are limited to observing only a single execution path. For each branch taken there exists an alternate path through the program that was not observed.

Ideally, a dynamic race detection tool would run multiple times with a variety of different inputs that cause every possible path to be executed and therefore analyzed by the dynamic tool. However, it is impractical to find enough inputs to cover all possible program paths. In a non-trivial application, finding enough inputs to cover even the majority of possible program behaviour is not practical. In practice, it is more likely that a dynamic tool will be run using a set of sample or test inputs used to mimic the common use cases of an application. Therefore, a fundamental limitation of dynamic race detection tools is their dependence on the input.

```
1  int len = 100;
2  double a[len]
3  #pragma omp target map(tofrom: a[0:len])
4  #pragma omp teams num_teams(2)
5      {
6          a[50]*=2.0;
7      }
```
Listing 2. DataRaceBench 116 with OpenMP offloaded to GPU

Races in Regions Offloaded to GPU. OpenMP offloading is becoming increasingly prevalent. Listing 2 shows a case from DataRaceBench which has a race in code offloaded to a GPU. All existing dynamic tools for OpenMP race detection are designed to be run on CPU and cannot analyze programs offloaded to GPU. A dynamic tool needs to consider extra details about the device to properly support GPU race detection. The hardware level structure of a GPU differs so dramatically from a CPU that applying a dynamic technique designed for CPU directly to GPU will almost certainly fail. Extending a dynamic OpenMP race detection tool to work for GPU would be a huge effort and to date no one has been successful in this area.

```
1  #pragma omp parallel for shared(A)
2  for(int i = 0; i < 10; i++) {
3      A[i] = i;
4      if (i == 1) { A[0] = 1; }
5  }
```
Listing 3. Race Depends on Number of Threads

Configuration-dependent Races. Another drawback of dynamic race detection tools is they generally have no knowledge
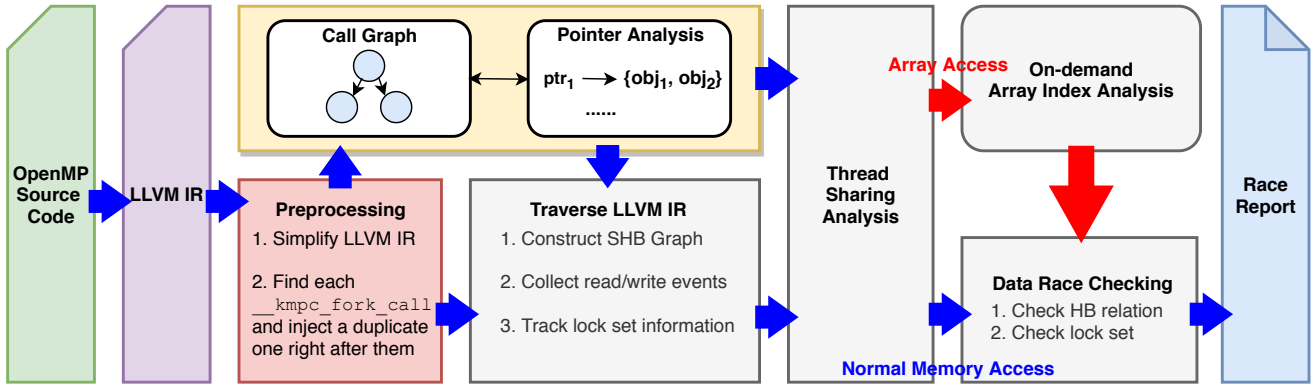
Fig. 1. An overview of OMPRACER.

of source level information. In the context of OpenMP race detection, this means that many dynamic tools may miss possible races because they lack higher level information about OpenMP. Consider Listing 3, in which there is a data race between the iterations of the loop where i equals 0 and 1. When this program is run with only two threads the race may not occur. The compiler may simply divide the loop iterations by the number of threads. In this case, iterations 0 through 4 would be executed on the first thread and iterations 5 through 9 would be executed on the second thread. When the iterations are split in this way, the two racing accesses in iterations 0 and 1 are executed by the same thread, meaning there is no race. The race is only exposed when the loop is split such that iterations 0 and 1 are executed on different threads. In Listing 3, depending on the OpenMP runtime scheduling, the program may be race free when run with up to 9 threads. However when run with 10 threads, iterations 0 and 1 will be scheduled on different threads and suddenly there is a data race.

Implications for Highly Parallel Systems. The types of races described above are more likely to be exposed on systems with a large number of threads. This can be particularly problematic if the machine used to test the application has fewer threads than the machine used to do production runs. A data race that occurs in production may not be observed on the machine used to test due to the smaller number of threads. This type of race can also be problematic over time as applications are run on new and more powerful machines. An application that was designed and tested on a 128 core machine may have run correctly for years, but when moved to a newer machine with more cores or even ported to run on a GPU suddenly experiences non-deterministic behaviour.

The static nature of OMPRACER allows us to address the aforementioned challenges faced by dynamic tools. Moreover, underpinned by our new advances in static analysis, OMPRACER is able to achieve both high performance and high accuracy, as we will show in the rest of this paper.

B. Race in miniAMR

OMPRACER has been used to detect a previously-unknown race in a real OpenMP application: miniAMR [14].

```
1  #pragma omp parallel default(shared) \
2                       private(i, j, k, bp)
3  {
4  for (in = 0; in < sorted_index[num_refine+1]; in++)
5  {
6    bp = &blocks[sorted_list[in].n];
7    for (i = 1; i <= x_block_size; i++)
8      for (j = 1; j <= y_block_size; j++)
9        for (k = 1; k <= z_block_size; k++)
10         work[i][j][k] =
11           (bp->array[var][i-1][j  ][k  ] +
12            bp->array[var][i  ][j-1][k  ] +
13            bp->array[var][i  ][j  ][k-1] +
14            bp->array[var][i  ][j  ][k  ] +
15            bp->array[var][i  ][j  ][k+1] +
16            bp->array[var][i  ][j+1][k  ] +
17            bp->array[var][i+1][j  ][k  ])/7.0;
18   for (i = 1; i <= x_block_size; i++)
19     for (j = 1; j <= y_block_size; j++)
20       for (k = 1; k <= z_block_size; k++)
21         bp->array[var][i][j][k] = work[i][j][k];
22  }
23 }
```

Listing 4. Data Race Discovered by OMPRacer

Listing 4 shows the OpenMP code containing the data race in miniAMR. The code begins with `#pragma omp parallel`, which specifies that every thread in the team will execute the entirety of the enclosed region. This means that the shared index variable `in` used in the outermost loop can be set to zero and incremented in parallel by multiple threads simultaneously. This is clearly a data race.

Data races are undefined behaviour and at high optimization levels, the compiler could potentially generate code that produces wrong results. However, it at first appears that this race simply causes some iterations of the loop to be executed more than once as `in` is reset to zero.

In reality this data race is more sinister. In fact, because of the race on `in` there is also a race on the private variable `bp`, or rather the data to which `bp` points to. Note that lines 11-17 read from `bp->array` and line 21 writes to `bp->array`. Although `bp` is a private variable, if `bp` were to point to the same location on two different threads, both threads would be reading and writing to the same `bp->array` in parallel. Each thread calculates `bp` at line 6 based on the value of `in`. As was shown above, the race on `in` makes it possible for

3

two different threads to have the same value of `in` at line 6, causing different threads to be assigned the same value for `bp`. The data race on `bp->array` will almost certainly lead to incorrect results.

OMPRACER was able to detect both the race on `in`, and the race on `bp->array`.

C. Races in CovidSim

OMPRACER has been used to detect and find three real races on CovidSim [15], [16] which have since been confirmed and fixed by the CovidSim developers. CovidSim is a microsimulation model developed by the MRC Centre for Global Infectious Disease Analysis hosted at Imperial College, London. CovidSim is used to model the spread of COVID-19 and the results generated by CovidSim have played a crucial role in shaping government policy on how to effectively address the COVID-19 pandemic. The importance of the CovidSim model has led to increased scrutiny on the code base. The developers use various tools, including Intel Inspector [7], to check for data races. The CovidSim project has also received support from GitHub and Microsoft who lent professional software engineers to review the code base. Additionally, since the project's public release on GitHub, developers all over the world have spent time trying to identify race conditions in the CovidSim model.

However, despite this heightened level of attention and scrutiny, OMPRACER was able to detect three new data races. The newly detected races are in unreachable code by default and only executed under a specialized configuration. These races had no effect on the current or previous results from modelling the spread of COVID-19, but the authors mention they plan to extend their models in the coming months to include features relying on the buggy code we detected. OMPRACER helped to resolve the problem in advance. Figure 2 shows the report for one of the races found by OMPRACER.

The CovidSim races show the power of OMPRACER as a static tool. As it is practically impossible to have dynamic tools to examine every possible program path with an exhaustive set of inputs and configurations, static tools have the unique ability to analyze the program in its entirety and find the bugs in uncommon code paths that might otherwise go unnoticed.

III. THE OMPRACER TECHNIQUE

Fig. 1 shows an overall architecture of OMPRACER, consisting of three primary components: ❶ a memory alias analysis, ❷ a comprehensive static modeling of OpenMP semantics, and ❸ a general race detection engine. These components work together to achieve scalable and precise race detection for OpenMP programs.

A. Memory Alias Analysis

One fundamental problem for static race detection tools is identifying shared memory that might be accessed concurrently by multiple threads. We use two types of analyses for this purpose: 1) a context- and field-sensitive pointer analysis and 2) an inter-procedural value flow analysis. These two

```
======== Found a race between:
line 215, column 3 in Update.cpp AND line 145, column 8 in Update.cpp
Shared variable:
/covim-sim/src/SetupModel:287
287| if (!(nEvents = (int*)calloc(1, sizeof(int))))...

Thread1: /covim-sim/src/Update.cpp
 213|
 214|        //increment the index of the infection event
>215|        (*nEvents)++;
 216| }
 217|
Stack Trace:
>>> DoInfect(int, double, int, int) [src/Sweep.cpp:717]
>>> RecordEvent(double, int, int, int, int) [src/Update.cpp:147]
Thread2: /covim-sim/src/Update.cpp
 143| if (P.DoRecordInfEvents)
 144| {
>145|        if (*nEvents < P.MaxInfEvents)
 146|        {
 147|                RecordEvent(t, ai, run, 0, tn); //...
Stack Trace:
>>> DoInfect(int, double, int, int) [src/Sweep.cpp:717]

The OpenMP Region Causing this race:
/covim-sim/src/Sweep.cpp:
>701|#pragma omp parallel for schedule(static,1) default(none) \
 702|        shared(t, run, P, StateT, Hosts, ts)
 703|    for (int j = 0; j < P.NumThreads; j++)
 704|    {
 705|            for (int k = 0; k < P.NumThreads; k++)
 706|            {
Gets called from:
>>> main
>>>    RunModel(int) [src/CovidSim.cpp:409]
```

Fig. 2. The terminal report for a real race detected in covid-sim

analyses work together to reason about memory aliases. The pointer analysis is a whole program analysis that computes the points-to set of every pointer in the program and is able to distinguish whether different memory accesses can refer to the shared piece of memory. However, the pointer analysis is *array index insensitive*, meaning that it cannot distinguish two memory accesses on the same array but at different indices (e.g., `A[i]` and `A[i+1]`). As array indexing is prevalent in OpenMP programs, apart from the pointer analysis, we also developed an inter-procedural value flow analysis to provide stronger ability to analyze array accesses. We introduce them as follows.

1) Pointer Analysis for OpenMP: In OMPRACER, we implement a context- and field-sensitive Andersen-style pointer analysis [17]. The pointer analysis offers two fundamental capabilities to our race detector: *a)* it computes the shared variables information for data race detection and *b)* it computes the complete call graph of the target program in the presence of function pointers, which is essential for building the global happens-before graph and for achieving high code coverage.

Our pointer analysis advances prior research with the following two contributions:

- In addition to providing support for non-OpenMP code, our pointer analysis is the only openly available framework that supports OpenMP programs. Other tools [18] fail to provide meaningful results because of unhandled OpenMP APIs.
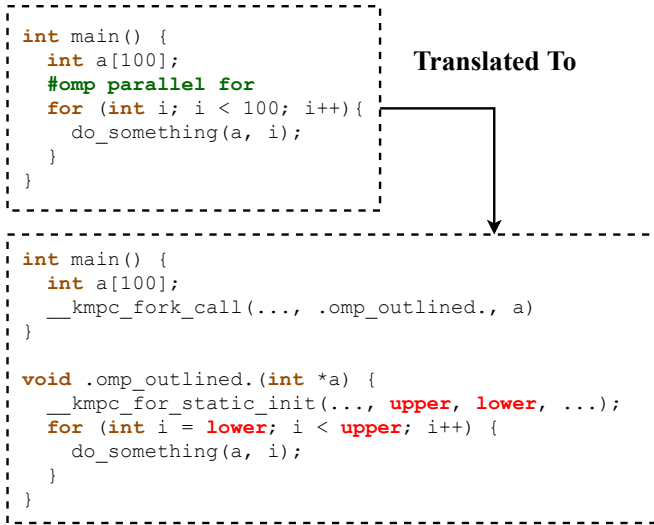- Instead of using conventional context sensitive pointer

Fig. 3. A (simplified) example of how an OpenMP parallel region is translated when compiled with clang/LLVM.
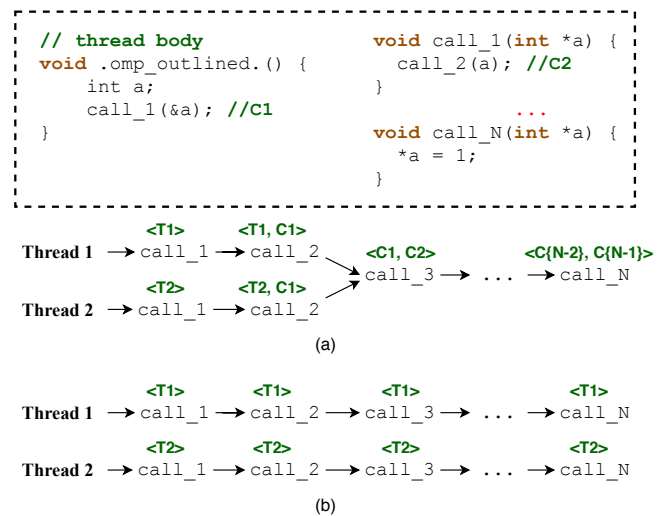

Fig. 4. A comparison between OpenMP region sensitivity and callsite sensitivity (with *k-limiting* set to 2). (a) 2-callsite sensitivity only maintains the most recent 2 call site to distinguish pointers; (b) OpenMP region sensitivity uses the logical thread ID to distinguish pointers.

analysis such as callsite sensitivity [19] and object sensitivity [20], we introduce a new type of context sensitivity, the OpenMP region sensitivity, that is designed specially for OpenMP programs to improve both the analysis precision and performance.

Understanding OpenMP APIs: To implement a pointer analysis framework that supports OpenMP programs, it is essential for the framework to understand the semantics of the OpenMP APIs and to model (potential) side effects correctly. When a call is made to the OpenMP runtime, pointer analysis should understand how data will be passed and processed by different OpenMP threads to compute the correct points-to information. Correctly modeling OpenMP requires an understanding of how high-level OpenMP features are translated by the compiler to LLVM IR. As the example shown in Fig. 3, to start an OpenMP `parallel for` region, the compiler first generates code to call the `__kmpc_fork_call` function to spawn OpenMP threads. The actual `for` loop body is encapsulated into `.omp_outlined.` functions, which are passed as callback functions internally invoked by the OpenMP runtime. To divide the `for` loop into disjoint sections that can be computed concurrently, `__kmpc_for_static_init` is invoked to split the loop statically as the OpenMP loop (implicitly) uses the static scheduling scheme.

As in Fig. 3, pointer analysis needs to understand how the `.omp_outlined.` function is executed and how the parameters are passed in to analyze shared objects between different OpenMP threads. It also needs to understand side effects caused by OpenMP runtime to abstract the behavior more accurately, e.g., `__kmp_omp_task_alloc` allocates objects and thus affects the points-to information. Our pointer analysis strictly follows the OpenMP standard and models most of the commonly used APIs. However, there are some less common OpenMP features that remain to be handled in the future (namely, `task` and `sections`).

OpenMP Region Sensitivity: Context sensitivity is essential for pointer analysis to improve the accuracy, it allows pointer analysis to distinguish the same static pointer when used by different callers. Apart from providing comprehensive support for OpenMP programs, we design and implement a new type of context sensitivity specially for OpenMP programs so that our pointer analysis is faster (and usually more accurate) when analyzing OpenMP programs than conventional approaches (e.g., callsite sensitivity). The key observations are

- To detect races in OpenMP parallel regions, the precision of the pointer analysis outside OpenMP parallel regions is less important. Performance can be improved by analyzing functions in a context-insensitive manner until an OpenMP parallel region is reached.
- For better scalability, conventional context-sensitive pointer analysis has to impose a *k-limiting* [21] to avoid the exponential complexity growth (e.g., only distinguish the most recent *k* callsites). This limitation could lead to unacceptable imprecision because of discarding critical contexts upon exceeding the *k-limiting* (e.g., where OpenMP regions are spawned). We will explain in the following paragraphs.

OpenMP region sensitivity overcomes the limitation by picking OpenMP threads' spawning sites as a context (normally a call to `__kmpc_fork_call`) and instead of updating the context whenever a new call is invoked, OpenMP region sensitivity only updates the context when new OpenMP threads are spawned. This allows users to handle nested OpenMP regions while imposing little overhead to the pointer analysis. Despite of the advantages that OpenMP region sensitivity has over conventional approaches when analyzing OpenMP programs, we make no claim that OpenMP Region sensitivity is superior in all other cases. Instead, we focus on showing how it improves OpenMP data race detection.

5

An important technical caveat specially for OpenMP programs is that one `__kmpc_fork_call` can create multiple threads. Thus, it is essential for pointer analysis to be able to distinguish different logical threads and provide meaningful data sharing information across threads. Our pointer analysis spawns two logical threads for each `__kmpc_fork_call`.

The difference between OpenMP region sensitivity and conventional callsite sensitivity is illustrated in Fig. 4. As shown in the figure, the thread body (i.e., the `.omp_outlined.()` function) allocates a *thread local* variable a. All the OpenMP threads then access their own copy of a in `call_N` and thus the example shown is *race-free*. The key to avoid reporting a false positive in the example is to recognize that the same pointer (pointer a) used in different OpenMP threads points to its own thread-local copy. 2-callsite sensitivity fails on this case because it only looks up the 2 most recent callsites. Thus, starting from `call_3`, the two pointers become indistinguishable because the thread spawn site is popped out as the *k-limiting* is exceeded. On the other hand, OpenMP region sensitivity reports no race on the example. Instead of using concrete callsite to distinguish pointers, OpenMP region sensitivity adopts the logical thread ID to distinguish pointers, thus, no matter how long the call chain is within an OpenMP region, the functions are analyzed under the context of different logical threads.

2) Interprocedural Value Flow Analysis: Our *interprocedural* value flow analysis is key to determining whether two array accesses can refer to the same location in memory. Our implementation reuses LLVM's Scalar Evolution (SCEV) analysis [22], a state-of-the-art symbolic analysis on scalar variables based on chains of recurrences [23]. However, SCEV is only intra-procedural, which does not work for array indices computed through multiple function calls. We made the following two major extensions on the original SCEV analysis:

- We extended SCEV to an on-demand context-sensitive inter-procedural analysis so that it is able to analyze array accesses across different functions.
- We built a fast constraint solver on top of the SCEV analysis to verify whether two array accesses can alias.

```
1  int arr[100];
2  for (i=1;i<100;i++) {
3    // SCEV: {%arr, +, 4}<for.body>
4    arr[i]=i;
5  }
```
Listing 5. A simple example on SCEV analysis

SCEV abstracts programs into SCEV expressions. For example in Listing 5, the index on array arr in the for loop is modeled by SCEV as `{%arr, +, 4}<for.body>`, which means that the address being accessed in the loop is increased by 4 (`sizeof(int)`) in each iteration. Note that although the example shown is simple, SCEV is capable of handing more complex loop structures in practice.

We utilize SCEV expressions and adapt them specially for OpenMP race detection. At a high level, OpenMP parallelizes for-loops by splitting the loop into chunks, each with its own induction variable which is calculated based in each thread's

TABLE I
LIST OF FEATURES SUPPORTED BY OMPRACER

Feature	Support	Feature	Support
omp parallel	✓	target	✓
omp for	✓	teams	✓
omp barrier	✓	simd	✓
master	✓	for simd	✓
single	✓	sections	✗*
reduction	✓	task	✗*
atomic	✓	taskwait	✗*
critical	✓	taskloop	✗*
threadprivate	✓	ordered	✗*

* This feature was incomplete at the time of writing. However there is no fundamental limitation preventing OMPRacer from supporting this feature.

id. Hence the array accesses within each sub-loops can all be viewed as:

$$f_{array}(tid) = f_{offset}(tid) + base$$

where $base$ is a constant computation for the base address of the array (independent of the induction variable), and $f(tid)$ is an offset computation related to the induction variable. To guarantee soundness, we assume that OpenMP will assign each iteration of the loop to a different logical thread. Then the alias checking for two array accesses $f_{array1}(tid1)$ and $f_{array2}(tid2)$, within a for-loop from low to $high$, can be formalized as:

$$\exists f_{array1}(tid1) = f_{array2}(tid2)$$
$$\text{where } tid1 \neq tid2 \text{ and } tid1, tid2 \in [low, high]$$

```
1  int i, j; double b[100][100];
2  #pragma omp parallel for private(j)
3  for (i=1;i<100;i++)
4    for (j=0;j<100;j++)
5      b[i][j]=b[i][j-1];
```
Listing 6. DataRaceBench Case No. 014

Listing 6 shows an example in DataRaceBench that contains a race on line 5 between `b[i][j]` and `b[i][j-1]`. Since j (line 4) starts from 0, it is possible for two different threads to access the same array element concurrently (e.g., between `b[0][99]` and `b[1][-1]`) due to array underflow. The simplified SCEV expression for `b[i][j]` and `b[i][j-1]` are:

$$f_{b[i][j]}(tid1) = j * 8 + tid1 * 800 + base$$
$$f_{b[i][j-1]}(tid2) = (j - 1) * 8 + tid2 * 800 + base$$

Here the $base$ is a constant value denoting b's base address and j is a constant range from 0 to 100. By omitting $base$ and applying the range of j, we can get the address range of `b[i][j]` is $[tid1 * 800, tid1 * 800 + 800]$ and the range of `b[i][j-1]` is $[tid2 * 800 - 8, tid2 * 800 + 792]$. Under the constraint $tid1 \neq tid2$, we can see that these two ranges can overlap, thus there exist potential races.

B. OpenMP Modeling

OpenMP parallel regions start with a call to `__kmpc_fork_call` functions. Our OpenMP modeling

uses the similar scheme as introduced in memory alias analysis by creating two logical threads for thread sharing and happens-before analysis. An important part for OpenMP modeling is that although variables are declared to be `private` or `shared` explicitly in OpenMP programs, we still need a whole program pointer analysis to compute data sharing information. In OMPRACER's implementation, we have modeled most APIs in OpenMP 5.0. Table I shows a summary of these features. In the following, we elaborate how we model some of the most common ones.

- **Master/Single** regions represent a section of code that should only be executed by a single thread. Events within a master or single region cannot race with themselves as they are guaranteed to happen on the same thread. However, they may race with events outside of the master/single region so they still need to be analyzed. This behaviour can be modelled by allowing only a single logical thread to execute a master/single region. By adding the events of a master or single region to only one logical thread, our analysis is able to detect possible data races involving events in a single region without falsely reporting races within the single region.

- **Reduction** allows for multiple values computed in parallel to be combined into a single result. An OpenMP developer generally specifies a variable that contains some piece of the result and an operation to combine each piece into a final value. The compiler then inserts code that handles the actual reduction. For simplicity, we assume the reduction code added by the compiler to be race free. Our tool is able to identify the reduction code and excludes it from analysis.

- **Critical** sections allow only a single thread to execute the specified block of code at a time. We model critical sections by treating them as if they were guarded by a lock. The start of a critical section is treated as acquiring a lock, and the end of the critical section is treated as releasing the same lock. We then use the lockset algorithm to determine if two events hold the same lock.

- **Atomic** regions in OpenMP are converted to atomic instructions at the IR level. Our tool excludes atomic instructions from analysis during race detection as they are guaranteed to be race free.

- **Barriers** represent a synchronization across threads in an OpenMP region. Barriers are modelled by adding a special barrier event to the static happens-before graph (described in Section III-C1). The first logical thread to encounter a barrier adds a barrier event and continues traversing the program. The second thread to encounter a barrier also adds a barrier event into its trace, but then must also create an bi-direction edge between its barrier event and the barrier event created by the first thread. This enforces a happens-before ordering between the two threads such that all events on both threads before the barrier must happen before all events after the barrier.

- **SIMD** in OpenMP signals that the compiler should trans-

late code to use SIMD instructions. We support SIMD for loops in OpenMP by disabling the SIMD translation and treating the loops as normal parallel for loops.

C. Race Detection Engine

1) Static Happens-Before Graph: Our race detection is based on a static happens-before (SHB) graph [24] extended with OpenMP semantics. The SHB graph is a directed graph whose nodes represent a *Program Event* and edges represent *Happens-Before relations*.

- **Events** represent the IR instructions relevant to data race detection, such as memory read/write, thread fork/join, and synchronization primitives. Table II shows a list of instructions and their corresponding events. Apart from normal read and write instructions, it is also necessary to model the common APIs for container classes in C++ standard libraries as read or write events, such as `vector`, `set`, `string`, etc.

- **Happens-Before Edges** connect the events to form a static representation of the program trace. There are two types of edges: 1) *Intra-thread HB edges* are implicit edges represented by unique ID of events. The events are created by traversing the program starting from the program entry, and each event is assigned with an unique ID at its creation. Therefore, the ID can be viewed as a "static timestamp" that implies its sequential orders with other events in the same thread. For example, event *A* with ID1 and event *B* with ID2 are two events in thread 1 and ID1 is smaller than ID2. Then there exists a Happens-Before relation from *A* to *B*. 2) *Inter-thread HB edges* are explicit edges that connect events in different threads. They are usually created by thread fork/join events or synchronization primitives such as signal/wait, memory barriers, etc. For example, thread 1 invokes a *thread fork* event, thus thread 2 is created. Then there will be an explicit edge from the thread fork event to the head of thread 2. In OpenMP programs, the thread fork/joins

are much simpler than normal concurrent programs, the majority of inter-thread HB edges are created by memory barriers.

The SHB Graph is a generic abstraction that fits into any multithreaded program in general. In OMPRACER, we identify a pair of `__kmpc_fork_call` (one created by OpenMP and the other injected by OMPRACER) and their corresponding `.omp_outlined.` function as the entry for each OpenMP region and start traversing the instructions inside. When function calls are encountered, we query the call graph generated by pointer analysis to find the instructions for that function. The final result of the SHB graph for an OpenMP region are two parallel sequence of events connected by some inter-thread HB edges.

2) Thread Sharing Analysis: Another core part for race detection is identifying all the shared memory locations and the memory access events on them. The shared memory is represented by abstract objects constructed by pointer analysis. For each abstract object, we maintain a *Writes Map* and a *Reads Map*, where we build a mapping between the static threads and the read/write access events on the abstract object.

Algorithm 1: ThreadSharingAnalysis

input : PTA - pointer analysis
CG - program call graph
output: $SharedObjects$ - a set of shared abstract objects

1   $threads \leftarrow$ two static threads of an OpenMP region
2   **for** $t \in threads$ **do**
3   **for** $inst \in t$ **do**
4       $pts \leftarrow$ getPointsToSet(*inst*);
5   **if** isWrite(*inst*) **then**
6   **for** $o \in pts$ **do**
7           $wmap \leftarrow$ getWritesMap(*o*);
8           $wmap \leftarrow (t, inst)$;
9   **else if** isRead(*inst*) **then**
10   **for** $o \in pts$ **do**
11          $rmap \leftarrow$ getReadsMap(*o*);
12          $rmap \leftarrow (t, inst)$;

13   FindSharedObjects();

Algorithm 1 traverses the OpenMP program IR instructions to collect the *Writes Map* and *Reads Map* for all the abstract objects we encountered (line 2-18). The maps not only record the correspondence between abstract objects and memory access instructions, but also reflect the the number of threads accessing each object (the keys of each map). Therefore we can use Algorithm 2 to identify shared abstract objects.

3) Connectivity Checking: The SHB graph abstracts the problem of checking happens-before relation of two events into a graph connectivity problem. To find if there's happens-before order between two events, a strawman approach is to perform a DFS (or BFS) starting from one event and vice

---

Algorithm 2: FindSharedObjects

input : $AbstractObjects$ - all abstract objects encountered
output: $SharedObjects$ - a set of shared abstract objects

1   **for** $o \in AbstractObjects$ **do**
2     $rmap \leftarrow$ getReadsMap(*o*)
3     $wmap \leftarrow$ getWritesMap(*o*)
4   **if** $wmap.size > 1$ **then**
5       $SharedObjects \leftarrow o$
6   **else if** $wmap.size = 1 \&\& rmap.size > 1$ **then**
7       $SharedObjects \leftarrow o$
8   **else if** $wmap.size = 1 \&\& rmap.size = 1$ **then**
9   **if** *writes and reads are on different thread* **then**
10         $SharedObjects \leftarrow o$

---

versa. This approach does not scale to realistic programs because it is common to have millions of events in the SHB graph. In our algorithm, we represent *intra-thread HB edges* as ordered event IDs, so that only the inter-thread HB relations require connectivity checking. We derive a *Synchronization Graph* from the SHB graph, which only consists of inter-thread edges and their corresponding nodes. Since the number of synchronizations is usually only small fraction compared to memory accesses, the size of the synchronization graph is significantly smaller than the SHB graph.

To check the connectivity of event *A* and event *B*, it is then sufficient to check the connectivity from A's immediate succeeding outgoing node to B's immediate preceding incoming node and from B's immediate succeeding outgoing node to A's immediate preceding incoming node. It is common for two events to have the same immediate succeeding outgoing node or the same immediate preceding incoming node, thus this method enables efficiently caching the connectivity results to optimize the performance.

4) Lock Set Tracking: OMPRACER uses a fairly straightforward approach to handle locks. Each read/write event in the SHB Graph will be associated with a *lockset* at its creation. The *lockset* is a global state that contains a list of abstract lock objects being hold when we reach the current event. Once a lock/unlock event is encountered, we query the pointer analysis to see which abstract lock objects the pointer may point to, and we update the *lockset* by pushing abstract objects into or popping the abstract object out of the *lockset*. The commonly used OpenMP instruction `omp critical` always acquires a pre-defined global lock by default.

5) Race Detection: Once all the essential data structures described above are constructed, the last step is to check each pair of memory access events on a shared abstract object:

1) check if there's at least one write event;
2) check if the two events are not connected on the SHB graph;
3) check if the two events do not share a common lock

(sharing at least one abstract lock object in their *lockset*).

6) Missed OpenMP Regions: To fully take advantage of pointer analysis in detecting shared objects, the baseline algorithm starts by looking for an entry function, like main, and builds/traverses the SHB graph by recursively visiting any called functions. This allows the tool to track the calling stack of any location visited and to track information about what objects are shared.

There is however a downside to this approach. There are some cases where static analysis may not be able to resolve what function is being called. This is especially true when there is unhandled external APIs. In these cases our tool may not realize that some functions are being called and may fail to analyze those functions. If those functions contain OpenMP regions, potential races could be missed.

To address this issue, we add a final step to scan for missed OpenMP regions. After running race detection from the entry function, we scan the code for any OpenMP region that has not been analyzed. If a missed OpenMP region is found, we set the function containing the missed OpenMP region as the entry function and run race detection again.

The challenge with this approach is reasoning about arguments passed into this function. If any of the arguments are pointers, or objects that contain pointers, we must make some assumptions about what they may alias with as pointer analysis lacks information about how this function was called. In order to produce meaningful results, we assume that any arguments passed in do not alias or overlap in anyway.

IV. EVALUATION

This section presents the evaluation of OMPRACER on DataRaceBench and the ECP proxy applications. We compare the results with three representative tools: Archer [3], ROMP [4], and LLOV [11]. Archer and ROMP are two state-of-the-art dynamic tools in use today and LLOV is a more recent static tool for OpenMP race detection.

All benchmarks used in our evaluation are written in either C or C++. However, because OMPRACER analyzes LLVM IR, OMPRACER is capable of detecting races in OpenMP programs written in Fortran as long as an LLVM front-end for Fortran is provided.

A. DataRaceBench

DataRaceBench is a collection of OpenMP microbenchmarks designed to evaluate and compare data race detection tools. As of v2.0, it contains a set of 116 benchmarks each with or without a specially designed data race that matches a misuse of certain OpenMP features, allowing evaluations to count the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) that each tool reports. Tools can then be compared using the following metrics.

- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- Accuracy = (TP + TN) / (TP + FP + TN + FN)
- Total Accuracy = (TP + TN) / (Total # of cases)

We introduce the Total Accuracy metric to give a complete picture of the accuracy across all 116 cases in DataRaceBench, including those cases on which some tools fail to run. While the Accuracy only reflects the cases on which each tool successfully runs, excluding cases where the tool fails to produce results. We use the statistics published on the DataRaceBench github repo [25] to compare our tool to Archer and ROMP in Table V. The results of dynamic tools can change from one run to the next as they may be sensitive to input, number of threads, and timing. So, the authors of DataRaceBench run each dynamic tool multiple times and give a min-max range for each metric. LLOV and OMPRACER have only a single value because as a static tool it is not dependent on any runtime behaviour and gives the same result every time.

The results in Table V show that OMPRACER significantly outperforms LLOV and even achieves a better result than the two dynamic tools in terms of the total accuracy. Both OMPRACER and LLOV have a very high recall, because both tools are static and they analyze the entire program and thus are unlikely to miss races. The false negatives cases in DataRaceBench are either due to unsupported features (task and sections), or due to compiler transformation. More specifically, there are two cases in DataRaceBench where the data race is removed by LLVM transformation passes. The simplified code pattern looks like the following code snippet:

```
1  int tmp; int A[100];
2  #pragma omp parallel for shared(A)
3  for(int i = 0; i < 100; i++) {
4    tmp = A[i];
5    A[i] = tmp;
6    // ===> Optimized as
7    // A[i] = A[i];
8  }
```

The above optimization is legitimate as data races are undefined behaviours (UB) in C/C++. And we kept those transformation passes as they simplify the LLVM IR and improve the accuracy of OMPRACER in general.

OMPRACER has a slightly lower precision than the dynamic tools and there are two main reasons for it: 1) To calculate the precision, the fail-to-run cases are excluded. Thus, OMPRACER, as the only tool that succeed on all the benchmarks, has a higher chance to fail as it evaluates more cases. In fact, OMPRACER has the highest total accuracy as in Table V, which considers fail-to-run cases as well; 2) There are some fundamental challenges for static analysis tool compared to dynamic tools that can cause the imprecision. For example, OMPRACER does not reason about branch conditions. When two threads can access the same memory location but there are branch conditions to ensure that only one branch is executed at runtime, OMPRACER will conservatively report a race. A dynamic tool would only observe one of the branches at runtime, so will not report the false positive. We considered those challenges for future improvements and they remain to be unsolved at the current version.

When compared to LLOV, OMPRACER shows much more promising result in almost every aspect (except for *Recall*, which is due to OMPRACER's much higher coverage on the

TABLE III
EVALUATION RESULTS ON LARGE REAL APPLICATIONS

Benchmark	#Instructions	LOC	#Regions	#Reads	#Writes	ROMP		Archer		OMPRACER	
						times (s)	#races	time (s)	#races	time (s)	#races
XSBench	5,617	6k	7	428	344	OOM	-	77	0	0.230	0
CoMD	6,454	11k	15	832	402	TO	-	49	0	0.739	4
Quicksilver	9,250	13k	1	1,166	806	CRASH	-	190	25	5.516	10
RSBench	10,637	6k	6	576	360	CRASH	-	138	0	0.225	0
miniFE	11,637	300k	21	866	776	CRASH	-	11	0	0.944	2
AMG	12,320	91k	2	2,534	792	CRASH	-	129	54	0.480	4
Lulesh	26,722	7k	30	3,534	1,542	CRASH	-	194	0	0.600	6
miniAMR	40,792	20k	36	7,054	1,214	CRASH	-	2,395	202	15	23
Kripke	90,288	700k	37	7,176	5,260	OOM	-	30	0	12	0
GROMACS	77,383,187	2,500k	82	4,980,655	2,173,035	CRASH	-	173	0	548.934	44

TABLE IV
PRECISION OF OMPRACER AND ARCHER ON COVIDSIM

Tool	#TP	#unknown	#race	Precision
OMPRACER	3	26	29	10%/100%
Archer	0	36(2)*	36(2)*	0%/100%

*Archer reports 36 races and 2 are unique.

TABLE V
DATARACEBENCH METRICS

Tools	Precision	Recall	Accuracy	Total Accuracy
Archer	0.98-0.98	0.90-0.91	0.94-0.95	0.90
ROMP	0.96-0.96	0.91-0.91	0.93-0.93	0.85
LLOV	0.83	0.94	0.86	0.63
OMPRACER	0.89	0.93	0.89	**0.91**

TABLE VI
DATARACEBENCH RESULTS

Tool	Correct (TP/TN)	Incorrect (FP/FN)	Mix	Error
Archer	104	6	1	5
ROMP	99	7	0	10
LLOV	73	12	0	31
OMPRACER	105	9	2	0

DataRaceBench).

The experiment results on DataRaceBench indicate that OMPRACER is the most powerful static OpenMP race detection tool so far. As shown in Table VI, OMPRACER is the only tool that is able to cover all cases in DataRaceBench and it reports the highest number of true positives and true negative among all the tools. Although OMPRACER is slightly worse than the dynamic tools in terms of precision, OMPRacer shows that a static tool is able to provide competitive results to dynamic OpenMP race detection tools. In the following section, we show that OMPRACER is also able to run much faster and achieve much higher code coverage when run on real-world application.

B. Evaluation on Real World Applications

We evaluated both the effectiveness and efficiency of OM-PRACER on a collection of ECP proxy applications (Table III) to show that OMPRACER is ready to be used in realistic OpenMP applications. These applications are all popular real-world applications that use OpenMP for parallelism. All the programs are non-trivial and the number of lines of code ranges from 6 thousands to over 2.5 million. All experiments were done on a desktop with an AMD Ryzen 9 3900 12-Core Processor and 32GB RAM in an Ubuntu 18.0.4 LTS docker container. For comparison we used Archer 2.0.0 built with spack [26], and the latest version of ROMP available from

their github repository. We could not run LLOV since it is not available.

Table III reports both the time spent on race detection as well as the number of reported races. Note that the time for compiling the project and generating the LLVM IR is not included. For OMPRACER, we first generated LLVM IR and then measured the time taken to perform race detection on that IR. For Archer and ROMP, we first generated instrumented binaries and then measured the time it took to run those instrumented binaries. If any benchmark took more than 30 minutes to complete, we report a time out (TO). If the process was killed for running out of memory, we reported OOM to represent out of memory. Unfortunately, despite our best efforts, we were unable to successfully run ROMP on any of the benchmarks tested, which also indicates the difficulty to develop practical tools on realistic programs.

In all but two cases, OMPRACER is able to analyze benchmarks much more quickly than Archer. This is because dynamic tools like Archer must execute the program and the time to run Archer on these benchmarks includes the time to run the program being analyzed in addition to the time to do the actual analysis. Static tools like OMPRACER do not have the overhead of running the tool and only run analysis.

The results in Table III reveal several interesting findings. First, OMPRACER is extremely fast. OMPRACER is able to analyze the majority of the benchmarks tested in under 1 second. Even on the largest benchmark, GROMACS, which contains over 77 million instructions to be analyzed, it completes within 10 minutes. Second, the number of races reported by OMPRACER is comparable to or fewer than the number of races reported by Archer, which is valuable as one would expect static tools always report more false positives than dynamic tools.

We are unable to determine the exact precision of either tool on the benchmarks in Table III as manual inspection to confirm the races reported on large programs requires extensive domain-specific knowledge. Without some ground truths like those given in dataracebench, it is difficult to know for sure which races are true or false positives in every case. A race we believe to be true under some input may actually be prevented by some path condition elsewhere in the program. Likewise, a race we believe to be a false positive may actually be possible under a very specific condition in a complex program. As such, we picked a single benchmark to study and report some comparison on precision. The results are shown in Table IV. OMPRACER reports 29 races in total and 3 of them have been confirmed by the developers. On the other hand, Archer reports two unconfirmed unique races on CovidSim, but misses the confirmed real races reported by OMPRACER. Archer misses the confirmed races OMPRACER reported because dynamic tools cannot analyze unreachable code paths. The remaining 26 unknown races reported by OMPRACER are all on the same variable as the unconfirmed races reported by Archer, but along different paths that Archer did not analyze.

Additionally, dynamic tools like Archer are only analyzing a single path through the program, OMPRACER is analyzing the entire program. This is why OMPRACER is slower than Archer when analyzing GROMACS. GROMACS is a huge application that consists of almost 100 sub modules. While Archer can only analyze a single path in one submodule, OMPRACER analyzes all 77 million IR instructions.

C. Limitations

As we can seen from the evaluation result, for benchmarks like Lulesh, CoMD, and miniFE, OMPRACER reported several races while Archer reported none. By manually checking those reported races, we confirmed they were all false positives, mainly because it is hard for static analysis to reason about the alias information for input-dependent data. Furthermore, since OMPRACER does not handle branch conditions, it will also discover races between accesses from two branches that cannot happen at the same time. The limitations above can be overcome by extending our analysis to include path-sensitivity.

Additionally, because OMPRACER runs analysis at compile time, it is unable to analyze precompiled libraries where the source code is not available at compile time. It is possible for OMPRACER to miss races in this case. Although OMPRACER internally models some commonly used libraries, such as the C++ standard libraries, to correctly analyze other precompiled binaries, the users will need to statically link the binaries with their executables.

V. RELATED WORK

Race detection has been considered as an important research topic for decades. Both static and dynamic algorithm has been proposed to tackle the problem. One of the key challenges is how to compute and represent *Happens-Before* Relationships.

Offset-span labeling [27], which is an online scheme that labels threads in a fork-join graph, labels each task with a vector of tuples. Vector Clock [28] records a clock for each thread in the system, and the virtual clock is increased upon every synchronization event. Two events are considered to be parallel if the two vector clock are not ordered. Flanagan et al [29] improve the vector clock algorithm by replacing heavyweight vector clocks with adaptive lightweight representation as they find the full generality of vector clocks is unnecessary in most cases.

Dynamic Race Detection Tools. Google's Thread Sanitizer [6], also known as TSAN, proposed a hybrid algorithm that uses both happens-before and lockset to detect data races. TSAN has been used to find hundreds of races in real world application. Helgrind [30] is a tool based on Valgrind [31]. Helgrind only detects happens-before relationships and it supports a subset o the dynamic annotations in TSAN. Intel's Inspector [7] is another dynamic data race detection tool that uses Intel PT [32] to trace the program. It uses a Concurrent Provenance Graph to record control, data and schedule dependencies.

Static Race Detection Tools. Chord [33], ECHO [24], D4 [34], RacerD [35], and SWORD [36] are static race detection tool on Java. Chord is based on object-sensitive, flow-insensitive alias analysis and escape analysis. RacerD leverages separation logic to detect races. It abandons the alias analysis and uses syntactic patterns to check alias information to achieve scalability. ECHO, SWROD, and D4 use field-sensitive but context-insensitive PTA. ECHO and D4 primarily focus on the incremental race detection, hence the SHB Graph design for handling function calls is different from OMPRACER.

LOCKSMITH [37], RELAY [38] are two static race detectors for C that both focus on precise lockset reasoning. RELAY uses a context-sensitive bottom-up algorithm leveraging the function summaries and symbolic analysis. LOCKSMITH uses a context-, flow-sensitive correlation analysis to infer the protection of locks and applies a sharing analysis to rule out thread local variables.

None of the static tools above support analyzing OpenMP programs, and only ECHO, D4, and SWORD try to reason about HB relations statically. OMPRACER, however, not only leverages the OpenMP region to achieve an efficient yet precise pointer analysis, but also extends the general abstraction of concurrent programs to support fast data race detection on OpenMP programs.

OpenMP Race Detection Tools. Various race detectors designed specifically for OpenMP programs have also been proposed. Archer [3] is a dynamic tool based TSAN. Archer achieves a low overhead by using a lightweight static analysis to only instrument code that cannot be verified to be race-free statically. However, Archer is based on TSAN and cannot detect logical OpenMP races. ROMP [4] is a recent dynamic tool that introduces a novel extension to offset-span labels [27] Atzeni et al. [5] proposed SWORD, an offline analysis tool. SWORD reduces the memory overhead by logging thread

traces into log files and analyze the log files in an offline manner. SWORD is also able to detect logical OpenMP races through an operational semantics of OpenMP [39]. LLOV [11] is a recent static tool built on Polly [40] that is capable of detecting OpenMP races and verifying that certain regions are race free. LLOV is also the only modern static tool capable of detecting races in OpenMP Fortran code.

VI. Conclusions

This paper introduces OMPRACER, a pure static analysis tool for OpenMP race detection. OMPRACER supports most of the commonly used OpenMP features. Unlike dynamic tools, OMPRACER is able to detect logical data races regardless of input and hardware configuration. It achieve nearly 100% code coverage and runs significantly faster than the state-of-the-art dynamic tools.

Our extensive evaluation on both DataRaceBench and real-world applications shows that static analysis can compete and in some cases outperform state-of-the-art dynamic OpenMP race detection tools. It also find a previously unknown data race in miniAMR.

Acknowledgment

References

[1] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 71–81.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[3] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 53–62.

[4] Y. Gu and J. Mellor-Crummey, "Dynamic data race detection for openmp programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 767–778.

[5] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, "Sword: A bounded memory-overhead detector of openmp data races in production runs," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 845–854.

[6] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[7] J. Thalheim, P. Bhatotia, and C. Fetzer, "Inspector: data provenance using intel processor trace (pt)," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 25–34.

[8] L. OpenWorks, "Helgrind: A data race detector, 2007."

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[10] F. Ye, M. Schordan, C. Liao, P.-H. Lin, I. Karlin, and V. Sarkar, "Using polyhedral analysis to verify openmp applications are data race free," in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2018, pp. 42–50.

[11] U. Bora, S. Das, P. Kureja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *arXiv preprint arXiv:1912.12189*, 2019.

[12] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompverify: polyhedral analysis for the openmp programmer," in *International Workshop on OpenMP*. Springer, 2011, pp. 37–53.

[13] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: a benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[14] A. Sasidharan and M. Snir, "Miniamr-a miniapp for adaptive mesh refinement," Tech. Rep., 2016.

[15] N. Ferguson, D. Laydon, G. Nedjati Gilani, N. Imai, K. Ainslie, M. Baguelin, S. Bhatia, A. Boonyasiri, Z. Cucunuba Perez, G. Cuomo-Dannenburg *et al.*, "Report 9: Impact of non-pharmaceutical interventions (npis) to reduce covid19 mortality and healthcare demand," 2020.

[16] [Online]. Available: https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/ScalarEvolution.cpp

[17] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[18] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[19] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004, pp. 131–144.

[20] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 17–30.

[21] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," *ACM Sigplan Notices*, vol. 29, no. 6, pp. 230–241, 1994.

[22] [Online]. Available: https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/ScalarEvolution.cpp

[23] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the international symposium on Symbolic and algebraic computation*, 1994, pp. 242–249.

[24] S. Zhan and J. Huang, "Echo: instantaneous in situ race detection in the ide," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 775–786.

[25] [Online]. Available: https://github.com/LLNL/dataracebench/wiki/Regression-metrics

[26] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[27] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 24–33.

[28] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Citeseer, 1988.

[29] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121–133, 2009.

[30] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–13.

[31] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[32] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015.

[33] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, 2006, pp. 308–319.

[34] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 359–373, 2018.

[35] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[36] Y. Li, B. Liu, and J. Huang, "Sword: A scalable whole program race detector for java," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 75–78.

[37] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 33, no. 1, pp. 1–55, 2011.

[38] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 205–214.

[39] S. Atzeni and G. Gopalakrishnan, "An operational semantic basis for building an openmp data race checker," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 395–404.

[40] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.