OpenRace: An Open Source Framework for Statically Detecting Data Races

Bradley Swain
Coderrect Inc.
College Station, TX
brad@coderrect.com

Bozhen Liu
Coderrect Inc.
College Station, TX
bozhen.liu@tamu.edu

Peiming Liu
Coderrect Inc.
College Station, TX
peiming@coderrect.com

Yanze Li
Coderrect Inc.
College Station, TX
yanze@coderrect.com

Addison Crump
Coderrect Inc.
College Station, TX
addison.crump@coderrect.com

Rohan Khera
Coderrect Inc.
College Station, TX
rohan.khera@coderrect.com

Jeff Huang
Coderrect Inc.
College Station, TX
jeff@coderrect.com

***Abstract*—Data races are a particularly nefarious type of bugs that can affect the correctness of parallel software. Data races are inherently non-deterministic, making them extremely challenging to detect and fix. High performance computing (HPC) applications are particularly vulnerable to data races as they are generally large complex applications involving massive levels of parallelism. Detecting data races in large scale, complex, and highly parallel applications can be nearly impossible without the help of domain specific race detection tools.**

We present the OpenRace framework, the only open source project aimed at providing the foundation needed to build a fast and precise static race detection tool for LLVM based languages. OpenRace is designed to be extensible and allow new parallel programming frameworks to be easily modelled without the need to write an entirely custom race detection engine, while also providing the flexibility to model complex domain specific features. We show the core components of the framework, and demonstrate how those components have been used to create a race detection tool for OpenMP. OpenMP is the standard choice for shared memory parallelism in the majority of HPC applications, and involves a number of complex features that can be challenging to model statically.

The OpenRace tool has thus far passed 149 of the 172 C/C++ cases in DataRaceBench version 1.3.2, outperforming all dynamic tools and ranking second place overall among the tools with results published by the DataRaceBench authors.

***Index Terms*—OpenMP, Data Race Detection, Static Analysis**

I. Introduction

Data races are among the most challenging types of bugs in software. Reasoning about complex timings and thread interactions is an inherently difficult problem. Data races are non-deterministic by nature, making them notoriously difficult to detect and debug. It is not uncommon for software developers to spend weeks or even months tracking down and understanding the root cause of a data race.

High performance computing (HPC) applications are particularly vulnerable to data races. HPC applications are generally highly complex and parallel systems at the forefront of hardware capabilities. OpenMP has been the de facto standard for on node parallelism in the HPC community for decades. OpenMP makes it easy to write highly parallel code that can take full advantage of accelerators such as GPUs. Although OpenMP hides most of the low level intricacies of writing parallel programs, there are no built-in safeguards against data races. It is left up to the application developer to ensure that OpenMP programs are data race free. Writing even simple parallel programs correctly can be challenging. Writing complex and highly parallel HPC applications can seem like a herculean task.

There are only a few production-ready data race detection tools for OpenMP, such as Intel Inspector [1], TSAN [2], Archer [3], ROMP [4], and Coderrect [5], most of which are based on dynamic analysis. Despite generating fewer false positives than static tools, dynamic analysis only analyzes a single executed path at runtime and can only report observed bugs. As a result, dynamic tools always miss hidden bugs. Additionally, repeatedly running dynamic tools with different configurations and inputs for higher coverage can be time consuming and exploring every possible program path with dynamic analysis is practically impossible. On the other hand, static analysis is perfectly suited for scanning the majority of a program. Static analysis tools analyze the entire source code directly, and therefore do not rely on specific hardware configurations, run time behaviour, or specific inputs. Critical bugs are often exposed only when some edge case path within a program is executed, and static analysis is well equipped to detect these kinds of bugs.

There has been a recent uptick in research in static race detection tools for OpenMP programs including DRACO [6], LLOV [7], and OMPRacer [8]. However, most of those works have focused on verification or analyzing a specific subdomain such as parallel loop dependencies. Moreover, few of these tools are open source and hence are unavailable to be extended for new features. Our previous experience on OMPRacer, a closed source static race detector, indicates the importance of extensibility of such tools as OpenMP is rapidly evolving and new features are added frequently into the standard. Thus, since OMPRacer, we have focused our efforts on developing an open source, highly extensible framework for static race detection.

In this paper, we present OpenRace [9], a framework for whole program static race detection in LLVM-based languages. OpenRace began as an open source redesign of the closed source Coderrect race detection tool with two main design goals:

- OpenRace is designed to be highly extensible. In the past we have developed different static race detection tools customized for different domains. Over time we have noticed redundant or similar code repeated across domains. In the context of race detection, pthread locks, C++ standard library RAII locks, and even OpenMP critical sections all behave nearly the same. The OpenRace framework aims to take advantage of this observation and create a framework that can be easily extended to support multiple domains.
- OpenRace is designed to allow complex domain specific features to be modelled accurately. We identify a set of abstract operations that are needed for race detection; Reads, Writes, Fork, Joins, Locks, and Unlocks. The OpenRace framework provides the core race detection analyses that operate only on these abstract operations. Then a specific framework can be modelled by mapping APIs to the abstract operations. See Table I for an example of how API calls are mapped to abstract operations.

OpenMP is the first programming model being modelled using the OpenRace framework, and although the work is on going, OpenRace has already been able to model many OpenMP features more accurately than the original closed source Coderrect tool. In DataRaceBench 1.3.2 [10], [11], the most recent version of the popular OpenMP data race benchmarks developed by LLNL, OpenRace has passed 149 of the 172 C/C++ cases, outperforming all dynamic tools and ranking second place overall among the tools with results published by the DataRaceBench authors. We expect the OpenRace framework to not only surpass the closed source Coderrect tool in OpenMP support, but eventually be extended to support pthreads, CUDA, std::threads, and a number of other domains.

To our knowledge, OpenRace is the only open source LLVM-based framework for static race detection. The code is fully open source and being actively developed. The project includes automated tests, coverage reports, and contribution guidelines aimed at ensuring quality and lowering the barrier for outside contributions. It has also been actively maintained with comprehensive documentation and rigorous code reviews.

OpenRace is publicly available at
https://github.com/coderrect-inc/OpenRace

In the rest of this paper, we first use an example to motivate our design choice. We then present the detailed design of OpenRace and its core implementation components, followed by our modeling of OpenMP features. We will also present our evaluation results of OpenRace on DataRaceBench, discuss related and future work.

II. Motivating Example

Developing a static data race detection tool that only handles the common cases may be relatively simple. Although the full OpenMP specification is huge, the majority of OpenMP usage likely falls into a small subset of the overall features. However, bugs often lie in mistakes made when venturing outside of the common patterns. Therefore, it is vital that data race detection engines are able to accurately model complex features and behaviours.

The OpenMP single directive can be used as an example to illustrate this point. The single directive tells the compiler that only one thread should execute a block of code. In Listing 1 the write to `global` is inside of a block marked with single. This ensures that only one thread will make this write, preventing a data race that would otherwise occur on this line.

```
#pragma omp parallel
{
  // Increment global counter by 1
  #pragma omp single
  { global++; }
}
```

Listing 1. Simple Single Race

A naive, but straightforward way to model single for race detection is to skip races in the same single region. However, this naive modeling is unsound. Listing 2 shows a hypothetical case where a programmer decided to wrap the update to `global` in a function and added `nowait` to prevent the code from blocking. However, in doing so, a race was introduced. Each call to `increment_global` results in a single thread updating a shared global variable. The use of `nowait` removes an implicit synchronization after the single, allowing both calls to be made in parallel. Lastly, although the single clause ensures that only one thread will execute the single region, the specification does not specify which thread will execute the single region. This means that one thread may execute the single region in the first call, while another thread may execute the single region in the second call in parallel, leading to a potential data race on `global`.

```
void increment_global() {
  // Added nowait to remove bottleneck
  #pragma omp single nowait
  { global++; }
}

// ...

#pragma omp parallel
{
   increment_global();
   increment_global();
}
```

Listing 2. Example showing an accidental data race using single

The naive model of OpenMP single will fail to detect the race in Listing 2 since at the source code level, there is only one single region updating global and it is considered to be race-free by the naive approach. The example shows how bugs can hide in the edge cases when users have misunderstanding on some features or look over some interactions in complex software.

TABLE I
MAP BETWEEN APIS TO ABSTRACT OPERATIONS

Pthread	OpenMP	Abstract Ops
pthread_create	kmpc_fork	Fork
pthread_join	*implicit*	Join
pthread_lock	kmpc_critical kmpc_set_lock	Lock
pthread_unlock	kmpc_end_critical kmpc_unset_lock	Unlock

The OpenRace framework has been built to provide the core analyses needed for data race detection, while also providing a base on which more complex domain-specific features can be accurately modelled.

III. THE OPENRACE FRAMEWORK

The OpenRace framework is built on top of the LLVM core libraries [12] and takes LLVM's intermediate representation (IR) as input. As a result, any language that can be converted to LLVM IR can be analyzed.

The OpenRace framework can be split into five stages: Preprocessing, Pointer Analysis, Function Summarization, Trace Building, and Analysis.

A. Preprocessing

We first preprocess the LLVM IR such that it is easier to be analyzed. The LLVM compiler framework transforms the IR by feeding it into a transformation pass pipeline. In OpenRace, we use a number of the passes provided by LLVM as well as some self-developed passes. The majority of these passes are standard compiler optimization passes not specific to any language or framework and are needed regardless of the specific code being analyzed. However, in some cases, preprocessing is used to model domain (language)-specific features, e.g., C++ virtual table.

For OpenMP programs, we apply an OpenMP specific pass to create fake duplicate calls to `kmpc_fork` to inform the analyzer that multiple threads might be created by a single call. Listing 3 shows a simplified version of the LLVM IR produced from the code shown in Listing 2 after preprocessing.

In general, preprocessing is only used for simplification of the IR. The majority of domain specific modeling happens in the later phases. Modeling language features during the preprocessing stage is used only as a last resort.

B. Pointer Analysis

The second step is running a custom whole program context- and field-sensitive Andersen-style pointer analysis. Pointer Analysis creates an abstract memory object for allocations in the program, and for each pointer in the program builds a set of abstract objects to which the pointer could potentially point to. These points-to sets can be used to find shared data across threads, and which instructions are potentially accessing that shared data.

The pointer analysis used in OpenRace employs a special type of custom context sensitivity, called *thread sensitivity*, that allows the analysis to detect shared pointers between threads while allowing less precise results for pointers within the same thread. Thread sensitivity is far too imprecise for most applications of pointer analysis, but that imprecision is what makes whole program pointer analysis feasible. By tracking only the information relevant to data race detection, i.e., shared data across threads, the pointer analysis can therefore be fast enough to scale to large and complex programs, while still being precise in the context of data race detection. The details of thread sensitivity are out of the scope of this paper, but have been discussed in a previously published work [8].

C. Function Summaries

Function summaries are built by taking LLVM IR functions and extracting only the abstract operations needed to detect data races. This is done through language models that map a specific LLVM instruction to an abstract operation, such as a Read or a Write. Irrelevant instructions, such as arithmetic operations, debug markers, or API calls that are known to not affect data races, can be ignored. An example of this mapping is shown in Figure 2

These summarized functions serve as a light weight wrapper around LLVM functions and can directly be used to build the static program trace in the next phase.

D. Static Program Trace

The main data structure used to detect races is the program trace. The program trace acts as a statically constructed per-thread trace of events that over approximate the runtime execution. The primary difference between a dynamic trace of events recorded at runtime, and our statically built trace are the program paths. A dynamic trace contains only one path through the program, while the static trace collapses all paths into a single list of events.

The program trace is built by recursively traversing a call graph generated by pointer analysis. As each function in the call graph is explored, the function is summarized as described in section III-C. A static event is recorded for each operation in the function summary. An event stores the underlying LLVM instruction, the type of the operation (Read, Write, Fork, etc), and the context in which the event was executed. For convenience, the events in the thread trace also distinguish between different types of call events. Calls to external functions are marked as extern calls and can be used as markers for later analyses. For non-extern calls, an event is added to the trace when the function is called, and when the function is returned from. These events are called BeginCall and EndCall respectively. These events can be used to reconstruct a callstack if needed by later analyses.

Listing 5 shows the resulting program trace for Listing 3. Each thread in the program trace lists an ID, entry function, and context for convenience. In reality, traces are more complex. The thread ID cannot always be determined accurately, and different events within the same thread may have different contexts. The program trace shown here is simplified to demonstrate the high-level concepts.

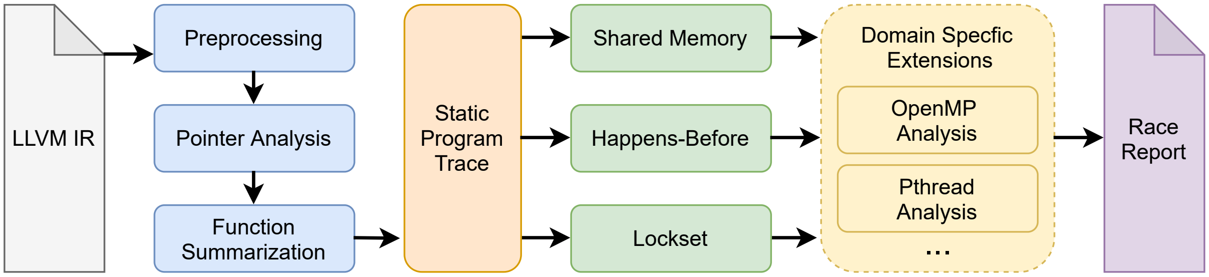


Fig. 1. An Overview of the OpenRace Framework.

```
define i32 @main(){
entry:
  call void @__kmpc_fork_call(@.omp_outlined.)
  call void @copied_kmpc_fork(@.omp_outlined.)
  ret i32 0
}

define void @.omp_outlined.() {
entry:
  call void @inc()
  call void @inc()
  ret void
}

define void @inc() {
entry:
  %0 = call i32 @__kmpc_global_thread_num()
  %1 = call i32 @__kmpc_single(i32 %0)
  %2 = icmp ne i32 %1, 0
  br i1 %2, label %then, label %end
then:
  %3 = load i32, i32* @global
  %inc = add nsw i32 %3, 1
  store i32 %inc, i32* @global
  call void @__kmpc_end_single(i32 %0)
  br label %omp_if.end
end:
  ret void
}
```

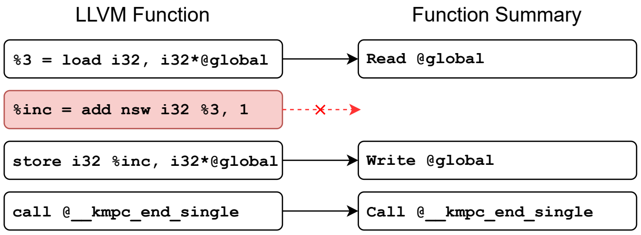
Listing 3. LLVM IR After Duplicate OpenMP Fork Preprocessing

```
@main summary
-------------
Fork
Fork

@.omp.outlined summary
----------------------
Call inc
Call inc

@inc summary
------------
Call kmcp_global_thread_num
Call kmpc_single
Read  @global
Write @global
Call kmpc_end_single
```

Listing 4. Function Summaries for the IR in Listing 3



```
%3 = load i32, i32*@global          Read @global
%inc = add nsw i32 %3, 1
store i32 %inc, i32*@global         Write @global
call @__kmpc_end_single             Call @__kmpc_end_single
```

Fig. 2. Translation from LLVM Functions to Function Summaries

E. Analysis

The Analysis phase contains the actual race detection logic. Three core analyses make up the default race detection check:

1) Shared Memory Analysis
2) Happens-Before Analysis
3) Lockset Analysis

Each analysis is designed to take the program trace as input and can then be queried for some property. Additional domain-specific analyses can be added to model features specific to a particular framework or language. There are a few different OpenMP specific analyses already included in the OpenRace project for modeling OpenMP specific features like single, lastprivate, sections, reduction, and array index analysis for parallel loops.

1) Shared Memory Analysis: Uses the points-to sets computed by pointer analysis to produce a list of all Read/Write and Write/Write pairs that access the same memory location from two different threads. The set of memory access pairs serve as the target of following analyses.

2) Happens Before Analysis: tracks which events must happen before other events. This property can be used to determine two events A and B may occur in parallel, by determining that neither A nor B have to happen before the other. The happens-before analysis creates a happens-before graph from the static trace, where each event is a node and an edge between two events represents a happens-before relationship. There are only three base cases where happens-before edges need to be added. 1) *Program order*, meaning events within the same thread must happen in order, dictates that there should be a happens-before edge between each event within a thread. 2) *Fork events* dictate that happens-before edges from a fork event to the first event on the spawned thread. 3) *Join events* dictate that happens-before edges from the last event on a thread to that thread's join event.

```
Thread 0
  entry:    main
  context: [main]
--------
0: Fork Thread 1 // new context [t1]
1: Fork Thread 2 // new context [t2]
2: Join Thread 1
3: Join Thread 2

Thread 1
  entry:   .omp_outlined.
  context: [main, t1]
--------
0:  BeginCall inc
1:  ExternCall kmpc_global_thread_num
2:  ExternCall kmpc_single
3:  Read  @global
4:  Write @global
5:  ExternCall kmpc_end_single
6:  EndCall inc
7:  BeginCall inc
8:  ExternCall kmpc_global_thread_num
9:  ExternCall kmpc_single
10: Read  @global
11: Write @global
12: ExternCall kmpc_end_single
13: EndCall inc

Thread 2
  entry:   .omp_outlined.
  context: [main, t2] // context changed
--------
// Identical event trace as Thread 1
0:  BeginCall inc
1:  ExternCall kmpc_global_thread_num
...
```

Listing 5. Thread Trace for the LLVM IR shown in Listing 3

3) Lockset Analysis: can be trivially implemented from the program trace. Each individual thread trace can be traversed. Starting with an empty lockset, simply add the corresponding lock object to the lockset when lock events are encountered, and remove the corresponding lock object from the lockset when an unlock event is encountered. Read or Write events will be associate with the lockset when they are traversed. During the race detection, we stop reporting on a pair when the intersection of the two locksets is non-empty.

IV. OpenMP Modeling

Modeling OpenMP features in the OpenRace framework is ongoing. The majority of the most common OpenMP features are already modelled. See Table II for a list of modelled features (green), partially modelled features (yellow), and features that have yet to be modeled.

Some OpenMP features can be implicitly modelled by the core OpenRace engine, but the majority need to be explicitly handled with custom analyses. We show a detailed example of how OpenMP single can be easily modeled in the OpenRace framework, and provide high level descriptions of how the remaining features are modeled.

TABLE II
Status of OpenMP Features Modeled in OpenRace

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

Entries with a dash indicate the feature is partially supported

A. OpenMP Single

The power of the OpenRace framework is the ability to write analyses that work on the static program trace, rather than LLVM IR directly. Detecting the race in Listing 2 using LLVM IR directly would be challenging, as at the IR level there appears to be only one single section. However, in Listing 5 each thread trace has two distinct single regions.

The general approach for modeling OpenMP single is similar to the naive approach, that is to not report races on events from within the same single region. The difference is in how single regions are identified as being the same. Rather than relying on LLVM IR, which cannot distinguish two calls to the same single region in `inc`, the static program trace can be used to accurately identify distinct single regions.

They key observation is that both thread traces are created from the same parallel region. This means that the number and ordering of single regions on each thread trace will be identical. The single regions within each thread trace can be numbered according to the order they are encountered, and these numbers can be used to determine if two single regions across threads in the same parallel region are the same.

First, single regions can be identified by recognizing a pairs of calls to `kmpc_single` and `kmpc_end_single`, marking the start and end of a single region, in the thread trace. Once a list of single regions has been identified, they can be numbered according to the order they are encountered within a given thread trace.

Next, given an event, the single region containing that event can be identified by checking that the event happens after the start of a single region and before the end of a single region. All events within a thread trace are numbered, so this check can be done as a simple comparison.

The full analysis, given a pair of events, must

1) Number single regions on each thread
2) Identify the single region containing each event
3) Compare the number assigned to each single region

If the numbers match, the events are within the same single region and no race should be reported.

Analyzing the static program trace provided by the OpenRace framework allows complex features to be accurately modelled in a way that is not possible on LLVM IR alone.

B. Other OpenMP Features

A high level description of how other OpenMP features are modelled in OpenRace is given below.

1) Parallel Region: The start of a parallel region is treated as a fork event. However, in the generated LLVM IR there will only be a single kmpc_fork call made to the OpenMP runtime. The runtime will then spawn a team of threads to execute the parallel region and wait for all threads to finish execution before returning.

To model the runtime spawning a team of threads, we include a preprocessing pass that duplicates each kmpc_fork call. This allows the later analyses to see that in reality multiple threads will execute the parallel region in parallel. To model the implicit joins at the end of each parallel region, a pair of join events is added after each pair of OpenMP fork events during thread trace construction.

2) Parallel Loops and Array Index Analysis: The core race detection engine is able to detect when two threads access the same memory location, but it cannot determine if array accesses inside of a parallel loop may overlap. We have developed a custom analysis that first detects if potentially racing accesses are inside a OpenMP parallel loop, and if those accesses are on a shared array. When both are true, a custom array index analysis is performed to determine if the two array accesses may overlap.

The array index analysis is intra-procedural and largely based on LLVM's Scalar Evolution (SCEV). When array access patterns are perfectly aligned for an access $a[i]$, as shown in Listing 6, we check whether array index (i.e., i in $a[i]$) is the loop index of OpenMP parallel loop (i.e., i below $\#pragma\ omp\ parallel\ for$). If so, we only report read/write races, e.g., the race includes the read of $a[i+1]$ and the write to $a[i]$. For multi-dimension accesses within nested loops, we also check that the nested indices are private (i.e., j in *#pragma omp parallel for private(j)*), which should be thread-safe.

```
#pragma omp parallel for
for (i = 1; i < N-1; i++) {
  a[i] = a[i+1] + 1;
}
```

Listing 6. Data race on an array within a parallel loop

3) Master: The master directive ensures that a section of code will be executed only by the "master" thread. Different master sections within the same parallel region are guaranteed to executed by the same thread, and so they can never race, however accesses within a master region may still race with accesses outside the master region. We model OpenMP master during thread trace construction by only placing events within a master section on the first thread in an OpenMP parallel region. This ensures events within master regions will never race with other events within a master region, but still allows for races with other code.

4) Section: OpenMP sections allow code to be split into sections which are each executed by a single thread. In clang, OpenMP sections are implemented as a switch statement inside of a parallel for loop where each case is a different section. This makes sections tricky to model for static analysis. Based on the guarantee that each section will only be executed by a single thread, we added an analysis to identify the begin and end of each section based on the switch target locations and do not report races where both accesses are within the same section.

5) Barrier: The barrier construct in OpenMP is a synchronization mechanism that forces all threads to wait until all threads in the team have reached the barrier. We model barriers as new type of Barrier synchronization event. When a barrier event is encountered during happens-before graph construction, a bidirectional happens-before edge is added between the barrier events on each thread encountering the barrier. The bidirectional edges enforce a happens-before relationship between all events on each thread before the barrier and all the events on each thread after the barrier.

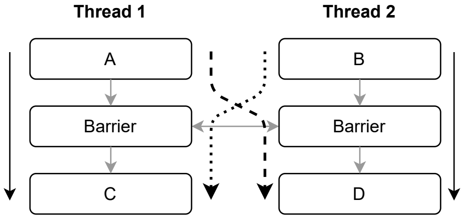


Fig. 3. Modeling of barriers in happens-before graph

Figure 3 shows an example of a Happens-Before graph with barriers on each thread. The barrier should enforce that A and B always happen before C and D. This can be confirmed observing a path from A to D, meaning A must happen before D. The same can be seen for B and C. This example shows how the bidirectional happens-before edge between barrier events accurately models the behaviour of a barrier.

6) Reduction: OpenMP allows for reduction clauses to be used with some directives. Reduction clauses allow for some final value to be computed in parallel from some value local to each thread. Reduction code is not written by the user, but directly generated by the compiler when a reduction clause is encountered. We make the assumption that reduction code will be race free as it is not written by the user. We model this by identifying each section of reduction code and not reporting races where both accesses are within the same reduction.

7) Critical: OpenMP critical ensures only one thread executes some region of code at a time. We model the start of each critical section as acquiring a global lock and the end of each critical section as releasing the global lock.

8) Task: OpenMP tasks are only partially supported at the time of writing. OpenMP tasks represent a section of code who's execution can be delayed and executed by any available thread in the team. This can be modelled by treating a task as a newly spawned thread. There are a few different task synchronization constructs to ensure that a task has completed execution, but by default a task can execute any time after it is created and before the end of the parallel region. We model this by tracking all task fork events and adding a corresponding join event at the end of each parallel region during thread

TABLE III
RESULTS ON DATARACEBENCH

Tool	Type	TP	FP	TN	FN	TSR	F1*
Archer	dynamic	63	1	80	17	0.936	0.819
Intel Inspector	dynamic	71	40	45	8	0.954	0.713
ROMP	dynamic	59	11	73	18	1.000	0.808
ThreadSanitizer	dynamic	64	1	84	15	0.955	0.838
Coderrect	static	72	2	85	9	0.977	0.907
OpenRace	static	71	8	78	10	0.971	0.862

trace construction. More complicated task synchronizations like taskwait, barriers, and more can be modelled in a similar way. Task synchronizations and other more complex task features such as taskgroup, taskloop, and task depend have yet to be fully modelled in OpenRace.

V. EVALUATION

The OpenRace framework is still a work in progress. Even so, the majority of the most commonly used OpenMP features have been modelled. Even with incomplete modeling, OpenRace has achieved very promising results on DataRaceBench.

A. *Results on DataRaceBench*

DataRaceBench is a suite of micro benchmarks designed to evaluate the effectiveness of OpenMP data race detection tools [10], [11]. DataRaceBench version 1.3.2 contains 172 C/C++ micro benchmarks and has published results for a variety of tools. The results provided by DataRaceBench [13] for Archer, Intel Inspector, ROMP, ThreadSanitizer, and Coderrect are shown in Table III along with the current results for OpenRace. Among these tools, only Coderrect and OpenRace are based on static analysis.

The number of true positives (TP) represents the number of benchmarks that did contain a race on which each tool reported a race. False positives (FP) represent cases where a benchmark did not contain a race but the tool did report a race. true negatives (TN), and false negatives (FN) follow similar definitions. The test support rate (TSR) shows what percentage of the benchmarks each tool was able to successfully run on. The adjusted F1 score ($F1*$) is calculated by multiplying the test support rate by the F1 score. The F1 score is equal to $2*(precision*recall)/(precision+recall)$ where $precision = TP/(TP+FP)$ and $recall = TP/(TP+FN)$.

Table III shows OpenRace has the second highest Adjusted F1 score at 0.862, just above Google's ThreadSanitizer and just below the closed source Coderrect tool.

Although OpenRace detects only one less true positive and 6 more false positives than Coderrect, there are a number of cases that can likely be passed after as the efforts to finish modeling OpenMP features continue. There are at least three task related cases and three simd related cases that OpenRace could potentially pass after support for those features has been added. Likewise, there are at least four task related and two offloading related false positive cases that could potentially be handled correctly in the near future.

Overall OpenRace is on track to surpass Coderrect's adjusted F1 score and have the best overall score on DataRaceBench. We plan to use the OpenRace framework as a base on which to build more powerful program analysis tools.

VI. LIMITATIONS

There are a number of limitations of static analysis that have yet to be addressed in the OpenRace framework. Two of the most severe limitations are described below.

A. *Path Sensitivity*

The path explosion problem is one of the limiting factors for whole program static analysis. The ability to analyze an entire program is a huge benefit of static analysis, but distinguishing between all possible paths of execution through any non-trivial program is impossible.

OpenRace avoids the path explosion problem by ignoring paths. By default, all paths are collapsed into a single static trace and analyzed together. After an initial set of races is detected, more complex analyses can attempt to try and determine if the path on which the race occurs is possible. Even so, infeasible paths are currently a common source of false positives. More sophisticated analyses may be used in the future to further reduce the number of false positives reported along infeasible paths.

B. *Indirect Function Calls*

Another fundamental limitation of static analysis is the inability to precisely determine where pointers could point at run time. This is the root cause of a number of problems in static analysis, of those the problem with the highest impact is resolving indirect function calls.

The majority of function calls are essentially hard-coded, and the function being called can be determined statically. However, a small fraction of functions are indirect, meaning the precise location being called is not determined until run time. Our whole program pointer analysis attempts resolve these indirect calls, however if any indirect calls are skipped or the target location cannot be resolved by pointer analysis a portion of the program is missed resulting in possible false negatives. If the target location is resolved incorrectly, there will likely be a large number of false positives. More work is needed to address indirect function calls.

VII. RELATED WORK

Research on data race detection has spanned multiple decades. Some of the earliest works formalizing happens-before was published by Leslie Lamport in the 1970's [14]. A number of different techniques and tools have been developed over the decades since.

Dynamic Tools: Two of the most widely used general purpose data race detection tools are Helgrind [15] and Google's ThreadSanitizer [2]. Both tools combine happens-before and lockset based dynamic analysis. Another popular general purpose race detection tool is Intel Inspector [1], which uses Intel PT [16] to record the program execution trace.

In addition to general purpose race detection tools, there are also a number of OpenMP specific dynamic tools. Archer [3] combines a lightweight static analysis pass with ThreadSanitizer annotations for the OpenMP run time. An extension to Archer, SWORD [17], has been proposed to add offline analysis capable of detecting more data races. Another recent dynamic tool, ROMP [4] uses an extension of offset-span labels [18] to efficiently track complex access orderings.

Static Tools: LLOV [7] is a recent static analysis tool for OpenMP. LLOV uses Polly [19], a polyhedral analysis framework for LLVM, to verify some OpenMP programs to be race free. LLOV supports both C/C++ and Fortran programs. Another OpenMP verification tool, DRACO [6] and OmpVerify [20] also use polyhedral analyses to verify some OpenMP programs to be race free. DRACO builds on the ROSE compiler [21] and focuses on verifying complex parallel loops. All of the above tools focus on verification of a subset of OpenMP programs. In general, verification tools work to prove a section of code as race free, where the OpenRace framework is geared towards reporting potential races.

The only other recent static analysis tool, that we are aware of, focusing on whole program static race detection for OpenMP is our previous work, OMPRacer [8] which has also been developed into the commercial Coderrect tool [5]. The core engine of OpenRace is also based on the techniques used in OMPRacer, but with additional abstraction and extensiblity allowing for more accurate modeling domain specific features.

VIII. Conclusion and Future Work

As we are completing OpenMP support in the OpenRace framework, there are a number of useful additions that can be built on top of OpenRace. Support for other domains like GPUs or pthreads can be developed. A light weight dynamic analysis can be added as a method of confirming race reports and reducing false positives. A smart ranking system could potentially be added to rank races by severity. The OpenRace framework is the foundation on which we plan to develop a practical data race detection tool for HPC and beyond.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Award Number DE-SC0020571. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] 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.

[2] 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.

[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] [Online]. Available: https://coderrect.com/

[6] 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.

[7] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, “Llov: A fast static data-race checker for openmp programs,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[8] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang, “Ompracer: A scalable and precise static race detector for openmp programs,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[9] [Online]. Available: https://github.com/coderrect-inc/OpenRace

[10] 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.

[11] G. Verma, Y. Shi, C. Liao, B. Chapman, and Y. Yan, “Enhancing dataracebench for evaluating data race detection tools,” in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2020, pp. 20–30.

[12] C. Lattner, “Llvm and clang: Next generation compiler technology,” in *The BSD conference*, vol. 5, 2008.

[13] [Online]. Available: https://github.com/LLNL/dataracebench/wiki/Metrics-Oct-2020

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

[15] 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.

[16] A. Kleen and B. Strong, “Intel processor trace on linux,” *Tracing Summit*, vol. 2015, 2015.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. Quinlan and C. Liao, “The rose source-to-source compiler infrastructure,” in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.