Does It Matter? - OMPSanitizer: An Impact Analyzer of Reported Data Races in OpenMP Programs

Wenwen Wang
University of Georgia
Athens, GA, USA

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA, USA

ABSTRACT

Data races are a primary source of concurrency bugs in parallel programs. Yet, debugging data races is not easy, even with a large amount of data race detection tools. In particular, there still exists a manually-intensive and time-consuming investigation process after data races are reported by existing race detection tools. To address this issue, we present OMPSanitizer in this paper. OMPSanitizer employs a novel and semantic-aware impact analysis mechanism to assess the potential impact of detected data races so that developers can focus on data races with a high probability to produce a harmful impact. This way, OMPSanitizer can remove the heavy debugging burden of data races from developers and simultaneously enhance the debugging efficiency. We have implemented OMPSanitizer based on the widely-used dynamic binary instrumentation infrastructure, Intel Pin. Our evaluation results on a broad range of OpenMP programs from the DataRaceBench benchmark suite and an ECP Proxy application demonstrate that OMPSanitizer can precisely report the impact of data races detected by existing race detectors, e.g., Helgrind and ThreadSanitizer. We believe OMPSanitizer will provide a new perspective on automating the debugging support for data races in OpenMP programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Parallel programming languages**; **Multithreading**.

KEYWORDS

Data race analysis, OpenMP, Happens-before orders, Concurrency bugs, Multithreading, Parallel program testing and debugging

ACM Reference Format:
Wenwen Wang and Pei-Hung Lin. 2021. Does It Matter? - OMPSanitizer: An Impact Analyzer of Reported Data Races in OpenMP Programs. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460379

1 INTRODUCTION

Parallel programming is notoriously difficult. Developers of parallel programs not only need to find efficient ways to identify parallelism opportunities and properly map them to low-level parallel hardware architectures, e.g., multi-core CPUs and GPUs, but also have to deal with nondeterministic concurrency bugs, such as data races and deadlocks. A previous study shows that it takes nearly 73 days on average to fix a concurrency bug [9]. Therefore, developing effective debugging tools is imperative to improve the programming productivity and lower the barrier of parallel programming.

Directive-based parallel programming models, e.g., OpenMP [12] and OpenACC [19], provide abstracted and portable *compiler directives* for developers to expose parallelism in sequential code. For example, OpenMP allows a programmer to parallelize a `for` loop by adding the simple "`#pragma omp parallel for`" directive before the loop. This significantly reduces the difficulty of parallel programming, as programmers can focus on the design of high-level parallel structures without worrying about the low-level implementation details. With more promising features added to support heterogeneous accelerators, e.g., GPUs and FPGAs, these programming models are growing increasingly popular as a powerful way to easily harness the power of parallel computing resources.

Unfortunately, however, developers still need to fight against concurrency bugs even with these easy-to-use parallel programming models. To give an example, it is quite easy to introduce a data race bug to an OpenMP program if a loop-carried dependence is overlooked when a loop is parallelized. Even worse, debugging such concurrency bugs is rather difficult due to their inherent *nondeterminism*, i.e., only triggered under specific execution interleavings. As a consequence, developers have to spend an enormous amount of time and effort to find and fix concurrency bugs during the development process.

As the major source of concurrency bugs, *data races* have a strong influence on the correctness, reliability, and portability of parallel programs. Essentially, a data race happens when two memory accesses from two different threads to a *shared* memory location are *not* ordered by any synchronization and at least one of the two accesses is a *write*. That is, the two accesses may exhibit different execution results due to the lack of necessary synchronization to determine their *happens-before* order. To facilitate the detection of data races, a large amount of research effort has been devoted to automatic data race detection [2, 3, 5, 6, 14, 18]. They typically employ a variety of static and dynamic program analysis techniques to figure out whether memory accesses issued by different threads may constitute data races. Notable data race detection tools include Helgrind [20], ThreadSanitizer [17], ARCHER [1], and etc.

Though these data race detection tools greatly mitigate the debugging effort of data races, developers still have to carefully investigate each data race reported by the detection tools. The reasons for this are multifold. First, a detection tool may miss some implicit synchronizations, e.g., user-defined adhoc synchronizations, and

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460379

thus reports *false alarms*. That is to say, a reported data race is probably not a true race. Second, a data race may be introduced intentionally with the purpose of achieving desired efficiency and scalability because of, for example, the high performance cost of synchronizations. In other words, the data race is a real race but it is *benign* rather than harmful. Third, even if a reported data race is a true positive and harmful, developers still need to further analyze the source code and execution traces of the program to understand the unexpected results caused by the race and find out its root cause so that the race can be fixed correctly. In short, there still exists a *manually-intensive* and *time-consuming* investigation process of data races after they are reported by existing detection tools. Therefore, creating automated tools to assist developers during this process is of paramount importance to reduce the debugging burden from developers and enhance the debugging efficiency.

Some previous research work attempts to address this problem partially by developing race analysis tools to classify benign and harmful data races [11] or data races and data race bugs [7]. However, several fundamental limitations of these tools inevitably hinder their practicability when adopted in real-world environments. First, they are tied closely to data race detection tools. Specifically, these analysis tools rely on customized extensions to existing data race detection tools to collect sufficient information during the race detection process. For example, most of these analysis tools need to precisely record and replay the execution interleaving of different threads in which a data race is reported. Obviously, this is impractical because, to the best of our knowledge, most extensively-used data race detection tools, e.g., Helgrind and ThreadSanitizer, do not have such capability. Moreover, extending existing race detection tools requires tremendous engineering effort and is not scalable as developers may adopt new detection tools to find more races. Second, a common assumption of these analysis tools is that an execution difference caused by the reversed happens-before order of the two memory accesses in a reported data race implies that the race is harmful. Though on the surface this assumption is reasonable, in-depth study shows that many benign data races can also lead to various execution differences, e.g., double-checked locking [15]. Therefore, it is *inaccurate* to determine the impact of a reported data race only based on the execution result. Last but not least, these analysis tools typically only provide developers with the analysis result of whether a reported race is worthy of further investigation or not. There is no additional information for developers to continue the analysis. As a result, developers have to start the manual analysis from the very beginning to collect necessary information.

To address the above limitations and provide an effective debugging tool for developers to analyze data races reported by existing detection tools, this paper presents OMPSanitizer, which aims to automatically analyze the *impact* of data races reported in OpenMP programs. OMPSanitizer adopts a *decoupled* system design and thus does *not* pose any requirement on existing data race detectors. Hence, it can be seamlessly integrated with existing popular race detectors, e.g., Helgrind and ThreadSanitizer, to assess the impact of data races reported by them. Another key feature that significantly distinguishes OMPSanitizer from existing race analysis tools is a novel and *semantic-aware* impact assessment mechanism, which is inspired by the following two observations.

(1) The happens-before order of the two shared memory accesses in a harmful data race can be *reversed*.
(2) The original and reversed happens-before orders of a harmful data race can *break* program semantic integrity.

Hence, rather than heuristically checking the execution differences between the original and reversed happens-before orders, OMPSanitizer conducts comprehensive program semantic analyses to determine the potential impact of each reported data race. For example, if a data race is concluded with the possibility to break the atomicity semantic, OMPSanitizer will rank it with a higher priority for manual investigation. Otherwise, the race will have a lower priority if OMPSanitizer considers it as benign. Moreover, OMPSanitizer provides detailed information about the analysis process, e.g., the program code and execution interleaving corresponding to the broken program semantic, so that developers can start manual investigation with less time and effort.

OMPSanitizer takes as input two memory access instructions in a data race and reports the potential impact of the race. In essence, OMPSanitizer comprises three major steps. For each data race, it firstly checks whether the reported happens-before order of the two memory accesses in the race can be reversed, then collects shared memory accesses relevant to the two accesses, and finally examines whether the race may destroy the semantic integrity between the two accesses and relevant accesses. More technical details will be presented in Section 3. We have implemented a prototype of OMPSanitizer based on Intel Pin [10], which is a dynamic binary instrumentation infrastructure. Evaluation results on a set of OpenMP programs from the DataRaceBench benchmark suite [8] and an ECP Proxy application [4] demonstrate that OMPSanitizer can correctly assess the impact of data races reported by existing race detection tools, including Helgrind and ThreadSanitizer.

In summary, this paper makes the following contributions:

- We present OMPSanitizer, which is an effective impact analyzer of data races in OpenMP programs. It can be combined with existing data race detection tools to further reduce from developers the burden of debugging data races.
- We propose a novel and semantic-aware mechanism to assess the potential impact of data races. The mechanism examines the possibility of a data race to break the integrity of program semantics through comprehensive program analyses.
- We implement a prototype of OMPSanitizer based on Intel Pin. The prototype can work on executable binaries without any requirement on program source code. We also overcome several technical challenges during the implementation.
- We conduct extensive experiments to evaluate the effectiveness of OMPSanitizer. The evaluation covers a broad range of OpenMP programs, and the results show that OMPSanitizer can successfully report the impact of detected data races.

The rest of this paper is organized as follows. Section 2 provides the necessary background knowledge of data races and OpenMP. Section 3 elaborates design details of OMPSanitizer. Section 4 describes our prototype implementation of OMPSanitizer. Section 5 shows experimental results. Section 6 discusses the potential limitations of OMPSanitizer. Section 7 presents related work. And Section 8 concludes the paper.

```
1 #pragma omp parallel for
2 for (i = 0; i < n; i++)
3   A[i] += B[i] + C[i];
```

(a) A data-race-free parallel loop

```
1 #pragma omp parallel for
2 for (i = 0; i < n; i++)
3   A[i] += A[i+1] + B[i];
```

(b) A parallel loop with a harmful race

Figure 1: OpenMP code examples w/ and w/o data race.

2 BACKGROUND AND MOTIVATION

In this section, we present the background knowledge about data races and OpenMP, as well as motivate our research work.

2.1 Data Races

In general, accesses to variables *shared* between different threads should be protected by synchronizations, e.g., mutex locks, to enforce the happens-before orders of the accesses and guarantee the atomic semantics. However, in practice, due to various reasons, developers may forget to add the required synchronizations. As a consequence, unprotected shared memory accesses may lead to incorrect or unexpected execution results. Such unprotected shared memory accesses are often called *data races*. Essentially, a data race comprises two unprotected shared memory accesses issued by two *different* threads and at least one of the two accesses is a *write* operation. In the remainder of this paper, we use $R_t(x)$ and $W_t(x)$ to respectively denote a read and write operation issued by thread t to shared variable x, and $\rightarrow$ to represent the *happens-before* relationship between two accesses. Hence, possible happens-before orders of a data race include $R_s(x) \rightarrow W_t(x)$, $W_s(x) \rightarrow R_t(x)$ and $W_s(x) \rightarrow W_t(x)$, where x is the shared variable and s and t are two different threads. Here, $R_s(x) \rightarrow W_t(x)$ means that $R_s(x)$ returns the value of x *immediately* before it is written by $W_t(x)$. Similarly, $W_s(x) \rightarrow R_t(x)$ means that $R_t(x)$ returns the value of x *immediately* after it is written by $W_s(x)$, and $W_s(x) \rightarrow W_t(x)$ means that the value of x written by $W_s(x)$ is *overwritten* by $W_t(x)$.

2.2 OpenMP

OpenMP is a *directive-based* parallel programming model. It allows programmers to write high-level "`#pragma`" compiler directives to exploit fine-grained parallelism in sequential code. This substantially simplifies parallel programming as programmers do not need to figure out how the parallelism in programs is mapped to low-level hardware parallel architectures, e.g., multicore CPUs. Since the version 4.0, OpenMP has been extended with the offloading feature, which allows developers to use corresponding directives to offload the execution of compute intensive parallel code regions to heterogeneous hardware accelerators, such as GPUs. OpenMP has received a considerable amount of attention and is becoming one of the dominant parallel programming models of many high-performance supercomputers.

Although OpenMP makes parallel programming much easier, it is still programmers' responsibility to identify the parallelism and *correctly* express them using the directives. That is, if a programmer fails to recognize data dependences between parallel code regions, a concurrency bug, e.g., data race, will be introduced. Figure 1 shows an example, where two `for` loops are parallelized in OpenMP. Here, the directive "`#pragma omp parallel for`" informs the OpenMP compiler and runtime that the iterations of the following `for` loop can be parallellized freely. Since there is no data dependence between different iterations in the first loop, the parallelized loop is *data-race-free*. But, the second loop has a loop-carried dependence, i.e., A[i] and A[i+1]. Therefore, a data race will be resulted in if this loop is simply parallelized using the same directive as the first one. Besides, this data race is harmful, as it will lead to incorrect execution results of A[i].

```
 1 int x, y = 0;
 2 #pragma omp parallel num_threads(2)
 3 {
 4    if (omp_get_thread_num() == 0) {
 5      x = 10;
 6      #pragma omp flush(x)
 7      y = 1;
 8    } else {
 9      while (y == 0);
10      if (x != 10)
11          printf("Error!");
12    }
13 }
```

Figure 2: An OpenMP code example with false positive and benign data races.

Motivation. Given the unexpected results caused by data races, many data race detectors have been developed to achieve automatic detection of data races. However, due to the lack of synchronization and semantic information of the target OpenMP program, existing race detectors may inevitably report *false positive* and/or *benign* data races. Figure 2 shows such an example. As shown in the figure, there are two shared variables, x and y. After analyses, an existing data race detector, e.g., Helgrind or ThreadSanitizer, will report two data races for this example. That is, the two accesses to x and y are reported as data races, respectively. In fact, with further manual investigation, we can conclude that the data race of the two accesses to y is a benign race, as it is used to implement a customized thread synchronization. Furthermore, the data race of the two accesses to x is a false positive, because their happens-before order is determined by the synchronization implemented by y.

Therefore, once a data race in an OpenMP program is reported by a race detector, developers often have to manually investigate it to confirm that it is indeed a *real* and *harmful* race, as it is quite common for race detectors to report false positives or benign races. Since this investigation process is manually intensive and requires developers' domain knowledge of the OpenMP program, it typically lasts for quite a long time. As a result, debugging a data race, even after it is reported by a data race detector, still requires significant engineering effort. OMPSanitizer aims at reducing the heavy debugging burden of data races from developers and enhancing the debugging efficiency by automatically analyzing the potential impact of reported data races. This allows developers to debug data races with much less manual effort. In next section, we will describe the design details of OMPSanitizer.

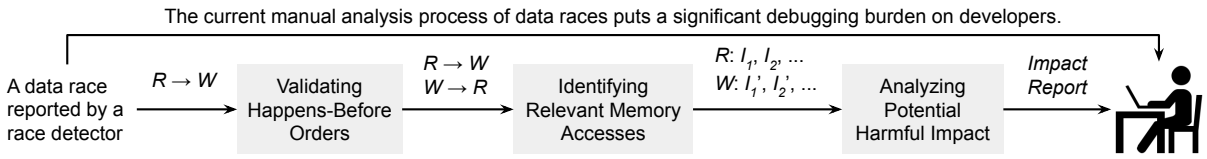

Figure 3: The high-level workflow of OMPSanitizer.

3 OMPSANITIZER

In this section, we first present an overview of how OMPSanitizer works, and then elaborate the design details.

3.1 System Overview

OMPSanitizer works as an enhancement of existing data race detection techniques. Specifically, it takes as input a data race reported in an OpenMP program by an existing race detector, e.g., Helgrind or ThreadSanitizer, and analyzes the potential impact of the data race. Based on the analysis result, OMPSanitizer categorizes the data race into one of the following three classes: false positive, benign, and harmful. In case the data race is harmful, OMPSanitizer provides additional information for developers to facilitate further manual analysis to find out the root cause of the bug. OMPSanitizer is designed to be *independent* from current data race detectors and also does *not* need any modification to the race detectors. Therefore, developers can use OMPSanitizer with their favorite race detectors. In addition, OMPSanitizer can also be integrated into existing race detectors through the provided plugin mechanism to offer seamless user experience. Figure 3 shows the high-level workflow of OMPSanitizer. As shown in the figure, the analysis in OMPSanitizer involves three major steps. We next explain each step in detail.

For each input data race with two shared memory accesses, indicated by two memory access instructions (i.e., R and W in the figure), OMPSanitizer firstly checks the happens-before orders of the two accesses. The purpose of this step is to validate that the happens-before order of the two accesses can be reversed. That is, one of the two accesses issued by two different threads can both happen before and after the other one. This step is necessary because a data race detector may miss user-defined synchronization operations between the two accesses and thus mistakenly reports them as a data race, as illustrated by the data race reported for the accesses to the shared variable x in Figure 2. If OMPSanitizer concludes that the happens-before order of these two memory accesses cannot be reversed, it will report the input data race as a false positive, which would provide a strong evidence for developers to ignore this data race report.

After validating the happens-before orders, OMPSanitizer next identifies memory access instructions that are relevant to the two accesses in the reported data race. Relevant accesses of an access A are defined as accesses that are conducted by the *same* thread as A and access the *same* memory location as A. The rational of collecting relevant accesses is that these accesses may form *semantic connections* with the accesses in the data race. For example, it is typically expected that two contiguous reads in a thread from the same shared memory location should return the same value. By identifying relevant memory accesses, OMPSanitizer can infer the potential semantic integrity of the accesses in the data race. OMPSanitizer leverages this semantic integrity information to determine the harmful impact of the data race through examining the possibility for the race to break the semantic integrity.

The last step of OMPSanitizer is to analyze the potential harmful impact of the input data race. To this end, OMPSanitizer firstly analyzes whether it is possible for the data race to break the semantic integrity inferred in the last step. OMPSanitizer achieves this by testing specific execution interleavings of the involved memory accesses. For example, suppose $R_s(x)$ and $W_t(x)$ are two accesses of the data race and $R_s(x)$ and $W_s(x)$ are inferred to have the atomic semantic. OMPSanitizer will test the following execution interleaving: $R_s(x) \rightarrow W_t(x) \rightarrow W_s(x)$, where the remote access $W_t(x)$ is scheduled between the two local accesses. If such an execution interleaving is possible, it means that the data race may break the atomic semantic between the two accesses. OMPSanitizer finally tracks the read operation to check whether the read memory value will produce unexpected side effects through an online taint analysis. This allows OMPSanitizer to more accurately determine the harmful impact of the data race. Note that OMPSanitizer only considers sides effects on *externally visible* results, e.g., program outputs or persistent disk files, as variances in internal execution states are not necessary to be harmful.

Now, let us use the example in Figure 2 to understand how OMPSanitizer analyzes the two data races. First, for the data race of x, OMPSanitizer will report it as a false positive. This is because the happens-before order of the two accesses is always $W(x) \rightarrow R(x)$, i.e., the read operation always return the value written by the write operation due to the synchronization realized by y. Second, for the data race of y, OMPSanitizer can observe both the happens-before orders of $R(y) \rightarrow W(y)$ and $W(y) \rightarrow R(y)$. For $W(y)$, there is no relevant access, as it is the only access to y in the thread. But for $R(y)$, OMPSanitizer will find out that relevant accesses come from the same memory access instruction because it is in the while loop. However, further analyses show that the read value only changes the iteration space of the loop and does not produce any side effect on the final execution result of the program. Therefore, OMPSanitizer reports the data race of y as a benign data race. With these information, it will be quite straightforward for developers to conclude that these two data races can be ignored without too much manual effort.

3.2 Validating Happens-Before Orders

The first step of OMPSanitizer is to validate the happens-before order of a data race detected in an OpenMP program. Typically, a data race is reported with two memory access instructions that access the same shared memory location. Hence, OMPSanitizer

starts from these two instructions to collect their happens-before orders by testing the OpenMP program comprehensively. Specifically, OMPSanitizer monitors the execution of the OpenMP program and dynamically examines whether one access can happen immediately before/after the other access. Note that OMPSanitizer also ensures that in the collected happens-before orders, the two instances of the two instructions access the same shared memory location. Here, we assume the input of the OpenMP program is available and the execution environment, such as the environment variable `OMP_NUM_THREADS`, is not changed compared to the race detection execution. We believe this assumption is reasonable and practical as a typical usage scenario of OMPSanitizer is to analyze data races after they are reported by a race detector on the same machine as the detector.

3.2.1 Exploring More Execution Interleavings. In general, when running an OpenMP program on a multicore processor, the execution interleaving of different threads is not deterministic across different executions. This depends not only on the inherent thread synchronizations and scheduling policy specified by the program, but also the implementation details of the OpenMP runtime. As a consequence, it is very likely that a happens-before order does not show up in a specific execution. To address this issue, OMPSanitizer intentionally injects random interference into the execution when testing an OpenMP program. This allows OMPSanitizer to explore more possible execution interleavings and collect as many happens-before orders as possible. To this end, OMPSanitizer carefully selects execution points to inject the interference. In particular, the points should have the potential to maximize the possibility of exposing more execution interleavings. Therefore, OMPSanitizer injects execution interference for the following execution events.

- **Shared Memory Accesses.** In general, most thread communications, including some user-defined synchronizations, are implemented through shared memory accesses. By disturbing the execution orders of shared memory accesses, OMPSanitizer can change the execution interleaving and experience more happens-before orders.
- **Synchronization Library/System Calls.** In addition to shared memory accesses, OpenMP runtimes may refer to low-level libraries, e.g., Pthreads, or system calls, to implement thread synchronizations. Hence, OMPSanitizer intercepts such library/system calls to inject execution interference for an enhanced testing coverage.
- **Thread Creations.** Depending on the scheduling policy, a newly created thread may lag behind the threads created earlier for the same parallel region, e.g., a parallelized `for` loop, or vice versa. Thus, to explore more execution interleavings, OMPSanitizer injects interference to parent/child threads after the child threads are successfully created.

OMPSanitizer realizes the execution interference by delaying the execution of the corresponding event with a *random* amount of time in nanoseconds. Obviously, it is impractical to inject an interference for every event, as an unacceptable performance overhead will be introduced. Therefore, OMPSanitizer installs a *random* interval for each type of events and injects the interference only when the end of the interval is reached. For example, OMPSanitizer injects an interference after a specific number of shared memory accesses have been executed. Besides, OMPSanitizer updates the interval each time after the interference is injected to increase the possibility of infrequent execution interleavings.

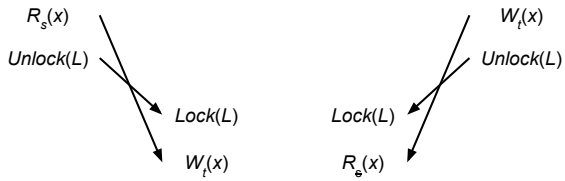

Figure 4: The happens-before order of the two shared memory accesses are dominated by the happens-before order of the lock/unlock synchronization.

3.2.2 Detecting Dominant Happens-Before Orders. After the above testing step, OMPSanitizer is able to tell whether the happens-before order of the two shared memory accesses in a reported data race can be reversed. However, even with the fact that the happens-before order can be reversed, it is still too early to conclude that the reported data race is not a false alarm. The reason is that the two accesses may be ordered by user-defined synchronizations, e.g., locks, which originally allow the happens-before order to be reversed. Therefore, OMPSanitizer needs to further check whether there exists a synchronization to enforce the happens-before order of the two accesses. This poses a great technical challenge on the design of OMPSanitizer, as it assumes no prior knowledge about the synchronizations of the target OpenMP program.

To address this challenge, a key insight of OMPSanitizer is that if there is a synchronization between the two memory accesses, the synchronization itself can also form a happens-before order. More importantly, this happens-before order always *dominates* the happens-before order of the two accesses. Figure 4 shows an example. As shown in the left side of figure, the happens-before order of the two shared memory accesses, i.e., $R_s(x) \rightarrow W_t(x)$, can be derived from the following happens-before orders: $R_s(x) \rightarrow Unlock(L)$, $UnLock(L) \rightarrow Lock(L)$, and $Lock(L) \rightarrow W_t(x)$, where $R_s(x) \rightarrow Unlock(L)$ and $Lock(L) \rightarrow W_t(x)$ are obtained from program orders. This is because happens-before orders are transitive. In this paper, if a happens-before order H_1 can be derived by another happens-before order H_2 and program orders, we call H_2 a *dominant happens-before order* of H_1. For example, in Figure 4, $UnLock(L) \rightarrow Lock(L)$ is a dominant happens-before order of $R_s(x) \rightarrow W_t(x)$. Therefore, the problem of checking the existence of synchronizations is transformed to detecting dominant happens-before orders.

To this end, OMPSanitizer invents a novel *window-based* approach to detect dominant happens-before orders based on the observation that a dominant happens-before order, if exists, is always formed *before* the corresponding happens-before order of the two shared memory accesses in a reported data race. Specifically, OMPSanitizer collects all happens-before orders formed between two different threads during the execution window of the two shared memory accesses in the reported data race. In addition to shared memory accesses, the happens-before orders collected by OMPSanitizer also include happens-before orders of two

Algorithm 1: Detecting Dominant Happens-Before Orders

Input: $thb = A_s \rightarrow B_t$ - The target happens-before order, where s and t are two different threads
$HBSet$ - The set of happens-before orders formed between A_s and B_t
Output: $DSet$ - Detected dominant happens-before orders

```
1  DSet = ∅
2  for hb = C_m → D_n ∈ HBSet do
3      if m ≠ s or TimeStamp(C_m) ≤ TimeStamp(A_s) then
4          continue
5      end
6      if n == t and TimeStamp(D_n) < TimeStamp(B_t) then
7          DSet = DSet ∪ {hb}
8          continue
9      end
10     DSet = DSet ∪ Recursive_Detection (thb, hb, HBSet)
11 end
12 return DSet
```

system calls issued by different threads. This is because system calls may also be used to implement thread synchronizations, e.g., `futex_wait/wake()`. It is worth pointing out that OMPSanitizer does not distinguish user-defined synchronizations and synchronizations implemented by OpenMP runtimes, such as OpenMP `critical` and `barrier` constructs, as both of them can be captured by OMPSanitizer uniformly through the detection of dominant happens-before orders.

At the end of the execution window, OMPSanitizer attempts to detect dominant happens-before orders. This is achieved by traverse all collected happens-before orders. Algorithm 1 illustrates the traversing process. The high-level idea is to examining the threads and timestamps of each collected happens-before order to see whether it is likely to be a dominant happens-before order. Note that dominant happens-before orders may involve more than two threads. For example, $A_s \rightarrow B_t$ may be dominated by $C_s \rightarrow D_m$ and $E_m \rightarrow F_t$, where s, m, t are three different threads. Therefore, Algorithm 1 recursively detects dominant happens-before orders consisting of multiple happens-before orders (Line 10).

For *every* occurrence of a happens-before order of the two shared memory accesses in a reported race, if a dominant happens-before order can always be detected for it, OMPSanitizer will consider the happens-before order is protected by a synchronization and classify it as a false positive. Otherwise, OMPSanitizer will continue the following analyses to determine its harmful impact.

3.3 Identifying Relevant Accesses

In this step, OMPSanitizer identifies relevant accesses for each access of the two memory accesses in a reported data race. This is inspired by the observation that harmful data races usually break semantic integrity of code regions in an OpenMP program. By identifying relevant accesses, OMPSanitizer can gather memory accesses that may have semantic connections with the accesses in the data race and further check whether the data race has a potential harmful impact on the semantic implication.

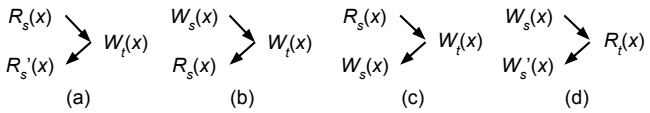

Figure 5: The happens-before orders of unserializable parallel execution interleavings. In each case, the left two accesses are interleaved with the right access from a different thread and thus their atomic semantic is not preserved.

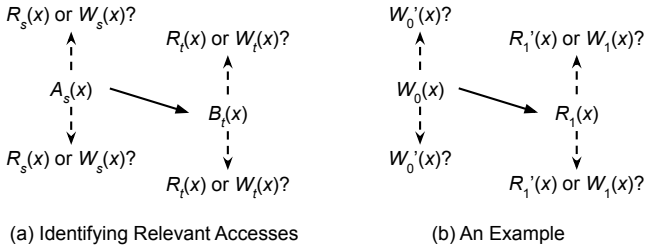

Figure 6: Bidirectional relevant access identification to find potential unserializable interleavings caused by data races.

Intuitively, a parallel execution interleaving of a correct OpenMP program should be *serializable*, which means the execution result of the parallel interleaving is equivalent to the result of a sequential execution of the program under the same input. However, due to the existence of data races, the serializability of an OpenMP program can be destroyed, leading to unexpected execution results. Figure 5 shows the happens-before orders of possible *unserializable* parallel execution interleavings between two different threads. Here, we assume the two memory accesses on the left side in each case have the *atomic* semantic and thus need to be executed atomically. Therefore, if any one of the left two memory accesses constitutes a data race with the access on the right side, the atomic semantic can be broken under the interleaving illustrated in the figure.

Figure 6(a) shows how OMPSanitizer identifies relevant accesses. Specifically, for each access of the two memory accesses in the race, e.g., $A_s(x)$ in the figure, where A is a read/write operation to x, OMPSanitizer attempts to identify accesses in thread s that also access x. OMPSanitizer mainly considers accesses that access the *same* shared variable for relevant accesses, as these accesses have a higher probability to form semantic connections in practice. Depending on the program order, a relevant access may happen *before* or *after* $A_s(x)$ in thread s. Therefore, the identification of relevant accesses in OMPSanitizer is *bidirectional*, as shown by the dotted arrows in the figure. In particular, when a memory access in the race is encountered in a thread during the execution, OMPSanitizer firstly look back at the access history of this thread to find accesses to the same shared variable. Then, after the execution of the memory access, OMPSanitizer continues to monitor the following accesses in this thread to collect future accesses to the shared variable. Additionally, if multiple relevant memory accesses are identified along one direction, OMPSanitizer will focus on the one that is most close to the access in the data race. The reason of this design choice is that the closest one has the *highest* possibility to compose semantic

implication with the access in the data race, compared to others on this direction.

An example of the above identification process is shown in Figure 6(b). In this example, $W_0(x)$ and $R_1(x)$ are the two shared memory accesses in the reported data race. For $R_1(x)$, OMPSanitizer seeks to identify both relevant read and write accesses to x before/after $R_1(x)$ in thread 1. This allows OMPSanitizer to further check whether $W_1(x)$ may break the potential atomic semantic between $R_1(x)$ and the identified relevant accesses. In a similar way, OMPSanitizer identifies relevant memory accesses for $W_0(x)$ in thread 0. But, as shown in the figure, OMPSanitizer only collects relevant write accesses to x before/after $W_0(x)$. This is because, even if there exists a relevant read access to x in thread 0, $R_1(x)$ will not break the potential atomic semantic between $W_0(x)$ and the read access, as demonstrated by Figure 5.

3.4 Analyzing Potential Harmful Impact

This section describes how OMPSanitizer analyzes the potential harmful impact of the reported data races, based on the information collected in previous steps.

OMPSanitizer firstly examines the possibility of the two memory accesses in the reported data race and the identified relevant memory accesses to form an unserializable execution interleaving, as shown in Figure 5, which is potentially harmful. OMPSanitizer achieves this using a technique similar to the one used to validate the happens-before order. For example, suppose $R_0(x)$ and $W_1(x)$ are two accesses in the reported data race and $R'_0(x)$ is a relevant access of $R_0(x)$ and happens after $R_0(x)$ in thread 0. To check the possibility of the unserializable interleaving $R_0(x) \rightarrow W_1(x) \rightarrow R'_0(x)$, OMPSanitizer individually checks each one of the two happens-before orders $R_0(x) \rightarrow W_1(x)$ and $W_1(x) \rightarrow R'_0(x)$. OMPSanitizer concludes that the unserializable interleaving is possible only if both of the two happens-before orders are confirmed to be possible.

If the unserializable interleaving is possible, OMPSanitizer then further checks the impact of the interleaving. This is done by online tracking the *read* operation involved in the interleaving. Recall that each unserializable interleaving in Figure 5 contains *at least one* read operation. In fact, it is often the value obtained by this read operation that propagates the impact of the data race to the following execution. Therefore, OMPSanitizer employs a classical dynamic taint analysis to check whether the read value, as well as the following reads from the same shared memory location, will affect *externally visible* execution results. Here, externally visible execution results refer to results that can be observed by the external world, e.g., results printed to standard output and results written to persistent storage. OMPSanitizer ignores intermediate execution results as it is quite common for OpenMP programs to have different intermediate execution results due to the inherent nondeterministic parallel execution interleavings. A reported data race is considered as harmful if the taint analysis demonstrates that the read value will affect externally visible execution results.

In case that no relevant access is identified for the two accesses in the reported data race, OMPSanitizer still checks the impact of the data race. If one of the two accesses in the race is a read operation, OMPSanitizer leverages the taint analysis described above to analyze its affection on externally visible execution results. Otherwise, if both of the two accesses are write operations, OMPSanitizer will monitor the value written by the last write and check whether it is read in the following execution and the read value is finally propagated to externally visible execution results. In either case, OMPSanitizer categorizes the reported data race as a harmful data race if the externally visible execution results are tainted. Otherwise, the reported data race is treated as a benign data race.

4 IMPLEMENTATION

We have implemented OMPSanitizer based on Intel Pin [10], which is a popular dynamic binary instrumentation infrastructure and has been widely used for whole program analyses, code profiling, and program testing. OMPSanitizer is implemented as a Pin tool to leverage the instrumentation facility provided by Pin. In this section, we report the obstacles we encountered during the implementation of OMPSanitizer as well as our solutions.

OMPSanitizer examines the address of a dynamically-executed instruction to determine whether it is a memory access in the reported data race. This poses the first challenge on our implementation, as the instruction addresses of an executable file may be changed across different executions due to different mapping addresses of *position-independent* code. To solve this issue, our implementation uses the *offset* instead of the exact address of an instruction in an executable file to uniquely recognize it, as the offset is fixed. The offset of an instruction can be calculated at runtime by subtracting the loaded address of the file from the instruction address. To obtain the loaded address of an executable file, We refer to the maps file of an execution, e.g., `/proc/16238/maps` on Linux, where `16238` is the process id of the execution.

As mentioned before, OMPSanitizer is composed of several stages and each stage has its own instrumentation requirement. For example, in order to validate the happens-before orders of the two memory accesses in a reported data race, the first stage of OMPSanitizer needs to monitor the execution of the two corresponding memory access instructions. In contrast, the second stage identifies relevant accesses of each access in the reported data race and therefore, it is required to keep the access history of each thread so that relevant accesses can be identified from the history. To address these different requirements on instruction instrumentation, our implementation creates various instrumentation strategies to accommodate the requirements in a single Pin tool. Specifically, we divide the execution into multiple phases and invoke a different instrumentation strategy in each phase. The instrumented code generated in the previous phase may also be flushed at the beginning of the current phase.

Since OMPSanitizer works at the binary code level, our implementation does not have any requirement on the availability of program source code. Besides, our implementation has no assumption on OpenMP synchronization constructs, which are typically implemented by OpenMP runtimes. But, in case the synchronization information is available, OMPSanitizer can also take advantage of them to enhance analysis efficiency and accuracy. This is similar to existing data race detection tools for OpenMP programs. For example, the LLVM OpenMP runtime exports OpenMP synchronizations to ThreadSanitizer through the compilation option `LIBOMP_TSAN_SUPPORT` to improve its detection accuracy.

Depending on the analysis results, OMPSanitizer files the analysis report with different information. For example, if a reported data race is classified as a false positive, OMPSanitizer will report the detailed validation results of happens-before orders and dominant happens-before orders if detected. In contrast, if the data race is considered as harmful, OMPSanitizer then provides the instructions whose semantic integrity is broken by the data race and the shared memory location accessed by the instructions. If the debugging information is available in the binaries, OMPSanitizer also maps the instructions back to source code and includes the source line information in the final report.

5 EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of OMPSanitizer. Our evaluation focuses on the research question whether OMPSanitizer is helpful to reduce the manual effort required to analyze data races reported by detection tools. To this end, we employ benchmark programs from DataRaceBench [8]. DataRaceBench is a benchmark suite that includes a comprehensive set of benchmark programs with and without data races. It was originally designed for systematic evaluation of data race detection tools. In addition to DataRaceBench, we also use an ECP proxy application miniAMR [4] to evaluate the effectiveness of OMPSanitizer on large-scale applications. All applications are compiled using LLVM (version 11.0.0) with the "-O3" optimization level. For DataRaceBench programs, we use the race detection results of Helgrind, while for the proxy applications, we use the detection results of ThreadSanitizer, as they are two widely-used data race detection tools. Since data races reported in application code are major concerns when developers analyze race detection results, the evaluation of OMPSanitizer covers all data races reported by both of the detection tools in application code, rather than the OpenMP runtime.

Our evaluation platform is equipped with a quad-core Intel E5-1620 v4 CPU at 3.5 GHz with the hyper-threading support and 32GB main memory. The operating system is Ubuntu 18.04.3 and the version of the Linux kernel is 4.15.0. The platform is occupied exclusively during the experiments to reduce the potential influence of random factors.

5.1 Analysis Results

Table 1 shows the analysis results of OMPSanitizer for the benchmark programs in DataRaceBench. Due to the space limitation, we cannot show the detailed results for all benchmark programs, as DataRaceBench contains more than 150 programs. Besides, we also omit benchmark programs for which Helgrind does not report any data races in the application code.

The first two columns of Table 1 shows the id of each program in DataRaceBench and whether the program contains a harmful data race, respectively. The third column shows the races reported by Helgrind for the benchmark programs. We use instruction addresses to represent the memory accesses in the reported data races. The races shown in this column also preserves the happens-before order of the two memory accesses when they are reported by Helgrind. The fourth, fifth and sixth columns (under H-B Order) show the validation results of happens-before orders. The seventh and eighth columns (under Relevant Accesses) present the identified relevant

```
1 int numThreads = 0;
2 #pragma omp parallel
3 {
4    if (omp_get_thread_num() == 0) {
5      numThreads = omp_get_num_threads();
6    } else {
7      printf("numThreads=%d\n", numThreads);
8    }
9 }
```

Figure 7: Benchmark #075 in DataRaceBench. The source code is simplified for demonstration.

accesses for the two memory accesses in the reported race. The impact analysis results are depicted in the last two columns, including the unserializable interleavings and the addresses of the instructions that produce externally visible side effects.

As shown in the table, for all data races reported in the evaluated benchmark programs, OMPSanitizer can correctly analyze their impact. Specifically, for all benchmarks that indeed have harmful data races, the analysis results of OMPSanitizer successfully conclude that some of the data races reported by Helgrind are harmful. For example, OMPSanitizer reports that two data races among the three data races detected by Helgrind for benchmark #027 are harmful, because they may form an unserializable execution interleaving and further lead to externally visible side effect. An interesting phenomenon we can observe from Table 1 is that even if no unseriablizable execution interleaving is identified for benchmark #075, OMPSanitizer is still able to categorize the two data races reported by Helgrind as harmful. Figure 7 depicts the source code of this benchmark. Here, `numThreads` is the shared variable and there are only two accesses to `numThreads`. The read access, i.e., `40064d`, comes from the line 7, while the write access, i.e., `400661`, is from the line 5. The impact analysis of OMPSanitizer correctly finds out that the read access can produce an immediate influence on the final execution result through the `printf` function call.

Another interesting observation of Table 1 is that a harmful unseriablization execution interleaving can be constructed by only two memory access instructions, as illustrated by benchmarks #011, #080, and #123. Further investigation shows that both of the two instructions actually involve two memory access operations. For example, the instruction `400748` in the benchmark #011 is `addl $0xffffffff, (%r14)`, which decreases the value stored in the shared variable `(%r14)` by one. Obviously, it includes a read operation, followed by a write operation. Moreover, this instruction is not prepended with a `LOCK` prefix, which is typically used to specify an atomic instruction in the x86-64 instruction set. Therefore, two instructions executed concurrently by two different threads can result in the following unserializable interleaving: $\underline{400748}\,(R_0) \rightarrow \underline{400748}\,(R_1) \rightarrow \underline{400748}\,(W_0) \rightarrow \underline{400748}\,(W_1)$, where the instructions with the same type of underlining are actually the same instruction.

From Table 1, we can also find that, for all benchmarks that do not have any harmful data races, OMPSanitizer is able to effectively determine that the data races reported by Helgrind are either false positives or benign. There are two cases of false positives. First, the

Program		Race Reported by Helgrind	H-B Order			Relevant Accesses		Potential Harmful Impact Analysis	
			O	R	D	First	Second	Unserializable Interleaving	Side Effect
005	yes	4008fc → 4007c3	✓	-	-	-	-	-	-
		4008fc → 400905	✓	✓	-	4008f7	40090a		
		4008fc → 40090a	✓	✓	-	4008f7	400905	4008f7 (R_0) → 40090a (W_1) → 4008fc (W_0)	4007c3
		40090a → 4008fc	✓	✓	-	400905	4008f7	400905 (R_0) → 4008fc (W_1) → 40090a (W_0)	4007d0
		40090a → 4008f7	✓	✓	-	400905	4008fc		
		40090a → 4007d0	✓	-	-	-	-	-	-
011	yes	400748 → 400684	✓	-	-	-	-	-	-
		400748 → 400748	✓	✓	-	4007ad	4007ad	400748 (R_0 & W_0) → 400748 (R_1 & W_1)	400684
		400748 → 4007a0	✓	✓	-	4007ad	4007a0	400748 (R_0 & W_0) → 4007a0 (R_1 & W_1)	
		4007a0 → 400748	✓	✓	-	4007a0	4007ad	4007a0 (R_0 & W_0) → 400748 (R_1 & W_1)	
		4007a0 → 4007a0	✓	✓	-	4007a0	4007a0	4007a0 (R_0 & W_0) → 4007a0 (R_1 & W_1)	
		4007a0 → 400684	✓	-	-	-	-	-	-
023	yes	400715 → 400656	✓	-	-	-	-	-	-
		400715 → 400644	✓	-	-	-	-	-	-
		400715 → 400715	✓	✓	-	400644	400644	400715 (W_0) → 400715 (W_1) → 400644 (R_0)	400644
027	yes	400756 → 400734	✓	-	-	-	-	-	-
		400756 → 400766	✓	✓	-	400734	-	400756 (W_0) → 400766 (W_1) → 400734 (R_0)	400734
		400766 → 400756	✓	✓	-	-	400734		
035	yes	400726 → 40070e	✓	✓	-	400759	-	400726 (W_0) → 40070e (R_1) → 400759 (W_0)	40065c
		400759 → 40070e	✓	✓	-	400726	-	400759 (W_0) → 40070e (R_1) → 400726 (W_0)	
053	no	40071a → 400673	✓	✓	✓	-	-	-	-
		400673 → 40071a	✓	✓	✓	-	-	-	-
074	yes	400930 → 400951	✓	✓	-	400951	400930	400930 (W_0) → 400930 (W_1) → 400951 (R_0)	4008b9
		400930 → 400930	✓	✓	-	400951	400951		
075	yes	40064d → 400661	✓	✓	-	-	-	-	40064d
		400661 → 40064d	✓	✓	-	-	-	-	
080	yes	4005c0 → 4005a4	✓	-	-	-	-	-	-
		4005c0 → 4005c0	✓	✓	-	-	-	4005c0 (R_0 & W_0) → 4005c0 (R_1 & W_1)	4005a4
084	yes	400822 → 400851	✓	-	-	-	-	-	-
		400822 → 400822	✓	✓	-	400851	400851	400822 (W_0) → 400822 (W_1) → 400851 (R_0)	400765
110	no	40089c → 4007ae	✓	-	-	-	-	-	-
		40089c → 400794	✓	-	-	-	-	-	-
		40089c → 40089c	✓	✓	✓	-	-	-	-
118	no	4008b8 → 4007eb	✓	-	-	-	-	-	-
		4008b8 → 4008e9	✓	✓	✓	-	-	-	-
		4008e9 → 4008b8	✓	✓	✓	-	-	-	-
120	no	4006d9 → 400711	✓	-	-	-	-	-	-
		400711 → 40068a	✓	-	-	-	-	-	-
123	yes	400716 → 4006f6	✓	-	-	-	-	-	-
		400716 → 400716	✓	✓	-	4006f6	4006f6	400716 (R_0 & W_0) → 400716 (R_1 & W_1)	4006f6
125	no	400649 → 40062e	✓	-	-	-	-	-	-
132	no	4009e3 → 400ab3	✓	-	-	-	-	-	-
		400aa6 → 400a64	✓	-	-	-	-	-	-
		400ab6 → 400a80	✓	-	-	-	-	-	-
143	no	400847 → 400804	✓	-	-	-	-	-	-
		400869 → 4007e0	✓	✓	-	-	4007e0	4007e0 (R_0) → 400869 (W_1) → 4007e0 (R_0)	-
		4007e0 → 400869	✓	✓	-	4007e0	-		
152	no	400978 → 4008cd	✓	-	-	-	-	-	-
		400978 → 400978	✓	✓	✓	-	-	-	-

Table 1: Analysis results of OMPSanitizer for benchmarks from the DataRaceBench suite. We use the data race detection results reported by Helgrind. "H-B Order:" happens-before order. "O:" the original happens-before order of the two memory accesses in the reported data race. "R:" the reversed happens-before order. "D:" dominant happens-before order. "First" and "Second" mean relevant accesses of the first and second accesses in the reported data race, respectively.

```
1 #pragma omp parallel default(shared) private(i, j, k, bp)
2 {
3    for (in = 0; in < sorted_index[num_refine+1]; in++) {
4      bp = &blocks[sorted_list[in].n];
5      for (i = 1; i <= x_block_size; i++)
6        for (j = 1; j <= y_block_size; j++)
7          for (k = 1; k <= z_block_size; k++)
8            work[i][j][k] = ...; // 0x525a0c
9      for (i = 1; i <= x_block_size; i++)
10       for (j = 1; j <= y_block_size; j++)
11         for (k = 1; k <= z_block_size; k++)
12           ... = work[i][j][k]; // 0x525ff5
13   }
14 }
```

Figure 8: The source code of miniAMR with data races, detected by ThreadSanitizer and confirmed by OMPSanitizer.

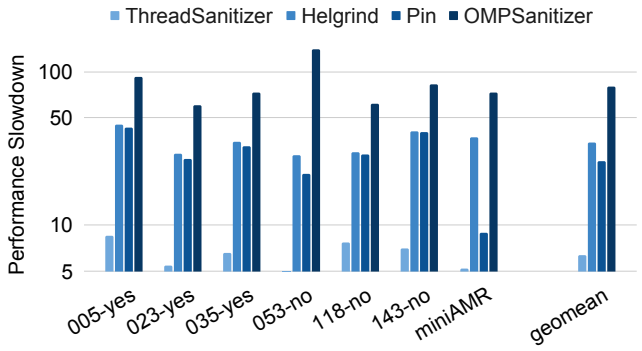

Figure 9: Performance slowdown of data race tools with the performance of native execution as the baseline.

happens-before order of the two memory accesses in a reported data race can not be reversed. Second, the happens-before order of the two memory accesses in the data race can be reversed, but OMPSanitizer discovers a dominant happens-before order for the data race. For example, all three data races detected by Helgrind for benchmark #118 are considered as false positives by OMPSanitizer, because the happens-before order of the first race cannot be reversed, while a dominant happens-before order is identified for the other two races, respectively. In addition to false positives, OMPSanitizer also identifies a benign data race as expected, i.e., the last two data races of the benchmark #143. This is because there is a user-defined synchronization in this benchmark. The synchronization employs a `while` loop to continuously read a shared variable in a thread until the variable is modified by another thread. This forms the unserializable interleaving shown in the table. However, OMPSanitizer still treats the data races benign as no impact on externally visible execution results is identified.

We next discuss the analysis result of the ECP proxy application, miniAMR. Among the 26 data races reported by ThreadSanitizer, OMPSanitizer successfully analyzes the impact of the data races and categorizes all of them harmful. We next use an example to explain the analysis results. Figure 8 shows a code snippet of miniAMR, where `0x525a0c` and `0x525ff5` are the addresses of the two instructions that access `work[i][j][k]`. ThreadSanitizer reports two data races for this code snippet: `0x525a0c` → `0x525a0c` and `0x525a0c` → `0x525ff5`. OMPSanitizer firstly confirms that the happens-before orders of the two races can be reversed. Then, OMPSanitizer identifies relevant accesses and finds out an unseriallizable interealving for the races: `0x525a0c` (W_0) → `0x525a0c` (W_1) → `0x525ff5` (R_0). Further analyses indicate that the values read from `work[i][j][k]` will be propagated to variables printed to standard output. As a result, both data races are considered as harmful by OMPSanitizer.

5.2 Analysis Efficiency

Since OMPSanitizer needs to dynamically instrument shared memory accesses at the binary code level, it inevitably introduces heavy performance overhead. Figure 9 shows the performance slowdown introduced by the data race tools discussed in this paper. The performance of native execution is used as the baseline. Here, we also include the performance of Pin without any instrumentation for reference. On average, as shown in the figure, the performance overhead of ThreadSanitizer, Helgrind, and OMPSanitizer are 6x, 35x, and 80x, respectively. Given that Pin itself introduces around 26x slowdown, we consider the performance overhead of OMPSanitizer is acceptable in practice, as OMPSanitizer is mainly used for offline OpenMP program debugging. In addition, enhancing the analysis efficiency of OMPSanitizer is part of our future work.

6 LIMITATIONS AND DISCUSSION

OMPSanitizer is designed to augment existing data race detectors by analyzing the impact of reported data races. Hence, enhancing the detection capability of existing detectors to discover more data races is *not* the goal of OMPSanitizer. In other words, if a data race escapes the detection of existing detectors because, for example, the race is problem-size specific, OMPSanitizer will not be able to analyze its impact. A feasible solution is to leverage multiple data race detectors to find as many data races as possible in an OpenMP program and then use OMPSanitizer to analyze the impact of each reported data race.

A potential concern of the testing approach in OMPSanitizer for exploring more execution interleavings through perturbing the execution is that it provides no guarantee in theory of a *complete* coverage of the entire execution interleaving space. Even though this is true, our argument is that such a complete coverage is actually unnecessary in practice, because of the following three reasons. First, OMPSanitizer takes as input data races reported by a race detector, and therefore the execution interleavings that OMPSanitizer wants to explore may have been experienced before by the race detector. Our experience on real-world race detectors shows that popular race detectors, including Helgrind and ThreadSanitizer, do *not* intentionally explore special execution interleavings. Second, even though the entire execution interleaving space is huge, we find that the set of the happens-before orders of an OpenMP program is usually limited. This is because happens-before orders imply data dependences, which are generally avoided in OpenMP programs to maximize parallelism. This also aligns with previous research about

testing concurrent programs [13, 21]. Finally, sophisticated execution interleavings are often quite rare in real OpenMP programs, as developers typically prefer to compose simple and intuitive synchronizations to ease the debugging burden. According to these three reasons, we believe the testing approach in OMPSanitizer is practical and can provide reliable results.

7 RELATED WORK

A large amount of research work has been devoted to detecting data races in OpenMP programs [2, 3, 5, 6, 14, 16, 18]. Based on the detection mechanisms, there are typically two types of race detection tools: static and dynamic. Static detection tools analyze program source code to discover data races, while dynamic tools detect data races by monitoring the execution of target programs. For example, LLOV [3] is a static data race detector based on LLVM, while ROMP [6] tracks accesses, access orderings, and mutual exclusion to detect data races. Different from these data race detection tools, OMPSanitizer mainly aims to reduce the manual effort required to investigate the data races reported by the tools.

Some data race analysis tools also attempt to classify data races. For example, iDNA [11] finds potentially harmful data races and Portend [7] classifies data race bugs. The general idea of these tools is to reverse the happens-before order of a detected race and then observe execution differences. A data race is classified as harmful or a bug if any execution difference is observed. These tools have two fundamental limitations. First, they need a heavy weight record-replay framework to record the detailed execution interleavings and states during the race detection process, which makes it hard to integrate them into existing race detectors, e.g., Helgrind and ThreadSanitizer. Second, an execution difference does *not* necessarily mean a data race is harmful, for example the race may be used to implement a thread synchronization. The results in [11] also confirm this limitation. In contrast, OMPSanitizer overcomes these limitations through a completely decoupled design from existing race detectors and a semantic-aware impact analysis of unserializable interleavings to determine whether a data race is harmful.

8 CONCLUSION

Even with a large amount of data race detection tools, debugging data races in OpenMP programs is still not easy. This is mainly because there still exists a manually-intensive and time-consuming investigation process after data races are detected by existing detection tools. To address this problem, we present OMPSanitizer in this paper. OMPSanitizer aims to reduce the debugging burden from OpenMP developers through a novel and effective impact analysis on reported data races. Through this analysis, OMPSanitizer can accurately classify a reported data race into different categories based on its impact. This significantly reduces the debugging burden and improves the debugging efficiency, as OpenMP developers can simply skip false positive and benign data races and focus only on harmful data races. We have implemented a prototype of OMPSanitizer based on Intel Pin. Experimental results on DataRaceBench and miniAMR show that OMPSanitizer can effectively analyze the impact of the data races reported by Helgrind and ThreadSanitizer and deliver promising analysis results.

ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers for their valuable feedback and comments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and LLNL-CONF-819184. This work is also supported in part by a faculty startup funding of the University of Georgia.

REFERENCES

[1] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 53–62. https://doi.org/10.1109/IPDPS.2016.68

[2] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn. 2018. SWORD: A Bounded Memory-Overhead Detector of OpenMP Data Races in Production Runs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 845–854. https://doi.org/10.1109/IPDPS.2018.00094

[3] Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. 2020. LLOV: A Fast Static Data-Race Checker for OpenMP Programs. *ACM Trans. Archit. Code Optim.* 17, 4, Article 35 (Dec. 2020), 26 pages. https://doi.org/10.1145/3418597

[4] Courtenay Vaughan. September 2020. miniAMR - Adaptive Mesh Refinement Mini-App. https://github.com/Mantevo/miniAMR.

[5] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[6] Yizi Gu and John Mellor-Crummey. 2018. Dynamic Data Race Detection for OpenMP Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 61, 12 pages. https://doi.org/10.1109/SC.2018.00064

[7] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 185–198. https://doi.org/10.1145/2150976.2150997

[8] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. 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* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/3126908.3126958

[9] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[11] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

[12] OpenMP Application Programming Interface. November 2020. Version 5.1. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[13] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[14] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery,

New York, NY, USA, 27–37. https://doi.org/10.1145/268998.266641

[15] Douglas C. Schmidt and Tim Harrison. 1997. *Double-Checked Locking*. Addison-Wesley Longman Publishing Co., Inc., USA, 363–375.

[16] Markus Schordan, Chunhua Liao, Pei-Hung Lin, and Ian Karlin. 2019. *Detecting Data-Races in High-Performance Computing*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[17] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[18] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. 2020. OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 54, 14 pages.

[19] The OpenACC® Application Programming Interface. November 2020. Version 3.1. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf.

[20] Valgrind User Manual. 2020. Helgrind: a thread error detector. https://www.valgrind.org/docs/manual/hg-manual.html.

[21] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. 2014. Localization of Concurrency Bugs Using Shared Memory Access Pairs. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 611–622. https://doi.org/10.1145/2642937.2642972