**SWORD: A Scalable Whole Program Race Detector for Java**

Yanze Li  
Texas A&M University  
yanzeli@tamu.edu

Bozhen Liu  
Texas A&M University  
april1989@tamu.edu

Jeff Huang  
Texas A&M University  
jeff@cse.tamu.edu

**Abstract**—We present the design and implementation of SWORD, a scalable and fully automated static data race detector for Java, implemented as a plugin in the Eclipse IDE. SWORD is the first whole program race detector that can scale to millions of lines of code in a few minutes while achieving good precision in practice. The cornerstone of SWORD is a new algorithm that judiciously combines points-to analysis and happens-before analysis efficiently, without losing precision. We have evaluated SWORD on an extensive collection of large-scale open source Java projects. Our results show that SWORD detects more races and reports fewer false positives than the state-of-art race detector, RacerD. Moreover, SWORD requires no human effort to annotate code regions as required by RacerD. SWORD also displays comprehensive bug traces and racing pair information on the GUI, which make debugging the races easier. A demo video is available at https://youtu.be/XQ0CBy7mMaY.

**Index Terms**—Data Race Detection, Whole Program Analysis, Static Analysis, IDE

I. INTRODUCTION

Concurrency bugs, especially data races, can be easily introduced but hard to detect and fix, yet the demand for multi-threaded programs is ever increasing with the development of multi-core hardware. Developers need fast and reliable tools to detect concurrency bugs. Most existing techniques, however, are designed for the late phases of software development, e.g., testing and production. This often renders it more expensive to fix a bug compared to detecting and fixing the bug in the programming phase, because the developers’ memory of the context for a bug may decrease with time.

We present SWORD, a Static Whole prOgram Race Detector in the form of an Eclipse plugin. SWORD is inspired by our prior work ECHO [1] and D4 [2], in which an incremental pointer analysis and a static happens-before (SHB) graph are proposed to detect data races incrementally. SWORD focuses on detecting data races at the whole program scale, thus it trims off the incremental analysis, and simplifies the structure of the SHB graph based on typical race patterns. As a result, SWORD detects data races in the IDE in a batch mode and provides a more detailed display, e.g., showing complete race traces in the GUI.

To make SWORD scalable and useful, we have solved following the engineering challenges:

1) We implement a flexible parallel design for SHB construction and race detection.

2) We avoid redundant storage and computation for the whole detection by optimizing the SHB design and leveraging typical race patterns.

3) We provide a user-friendly GUI with comprehensive traces for developers to utilize.

Despite the enormous body of existing race detection research, there is a surprising lack of out-of-box static race detectors open-sourced for developers in real-world projects. For Java programs, we have known Chord [3] and RacerD [4]. Chord is a project dated back to 2006 and is no longer maintained. It is hard to compile and run it on current Java projects. RacerD is a compositional race detector that relies on developer annotations to detect bugs in code segments. We evaluated both SWORD and RacerD, and found that SWORD has comparable performance with RacerD for most benchmarks but it provides more precise detection results at the whole program level and requires no human effort. This makes SWORD competitive for a whole program analysis at the code review phase.

The source code of SWORD is publicly available at https://github.com/parasol-aser/SWORD

II. BACKGROUND

RacerD [4] is an industrial-strength static race detector for Java programs developed by Facebook, and can scale to millions of lines of code in a few minutes. Unlike most race detectors analyzing a whole program, RacerD only partially checks the source project. To achieve great scalability, developers of RacerD made the following design choices:

1) Do compositional analysis instead of whole program analysis

2) Replace points-to analysis with an aggressive ownership analysis [4].

3) Do not reason about the interleaving between threads.

The compositional design of RacerD also faces several challenges. Since RacerD does not reason about thread interleavings, it requires a lightweight annotation library to provide extra information. Otherwise, it can miss real data races. For projects without normative annotations, it requires extra human effort to annotate. In principle, RacerD is neither sound nor complete. Due to the lack of alias information, RacerD only checks syntactically identical access paths, which makes it unsound and may introduce many false negatives in the whole
program. Due to the lack of a precise call graph, RacerD may also report false positives containing infeasible memory accesses that can never be reached by the program. In practice, false positives are especially distracting to developers from fixing real bugs.

III. OVERVIEW

An overview of SWORD’s architecture design is shown in Figure 1a. After users import their projects, SWORD starts from an entry method (e.g., main). It traverses all the reachable methods in the call graph in order to record the events in the SHB graph. Then, SWORD checks all the shared memory accesses to find if there exists any race. Users are allowed to exclude certain libraries and define arbitrary entries to customize the analysis.

SWORD visualizes the detected races in the IDE in a well-organized way. Figure 1b shows two types of views provided by SWORD for developers to review the results. The bottom view shows the race relation between memory accesses. Its left column shows all memory writes in reported races. After clicking at each write access, the right column will display all the reads and writes racing with the write we click. The listed memory accesses can be expanded to view its full trace to help developers analyze the bug. The top right view lists all the data race pairs. In the source file editor, a bug icon is also attached to each racy statement.

IV. IMPLEMENTATION

The implementation of SWORD is based on the Eclipse IDE [5], WALA [6] and Akka [7]. SWORD takes the bytecode or Eclipse AST as inputs, and uses WALA to transform the inputs into the SSA form IR. The analysis contains three main components: points-to analysis, SHB graph construction, and race detection on the SHB graph.

### A. Points-to Analysis

In SWORD, we use 0-CFA in WALA to perform a context-, flow-insensitive and field-sensitive on-the-fly points-to analysis (PTA), so that we can obtain both call graph and pointer-assignment graph (PAG) where each variable node associates with a points-to set. The reason of choosing context-insensitive analysis is that the PTA sometimes takes more time than race detection when analyzing large programs. Even changing the context to 1-call-site, which is usually insufficient to filter out most false positives, makes the simplest test case 2X slower. Considering such a large performance cost, the accuracy improvement brought by context-sensitivity is relatively small.

### B. SHB Graph Construction

The SHB graph is critical in SWORD. It augments the call graph and points-to information in previous steps with directed edges to indicate the happens-before relation between nodes. Subgraphs are connected by edges indicating the inter-thread happens-before relations. However,
repeated method calls can introduce duplicate nodes from the same methods into subgraphs and make the SHB graph large. This will significantly increase the storage overhead and make our detection inefficient. To mitigate this issue, we represent both methods and threads as subgraphs. Any call to a method is translated to a happens-before edge from the invoke node to its subgraph, and the subgraph is associated with a list of tids to record the threads that has invoked this method. Moreover, since the threads generated in a loop always have the same structure in the SHB graph, we do not unroll those loop but simply label them as threads created by loops. This further simplifies the SHB graph and avoids some redundant computations during the detection phase.

We refer our readers to [1] for a detailed description of the SHB graph construction. The construction process is further parallelized to speed up the tool.

C. Race Detection Engine

SWORD detects races in three steps: 1) Find shared abstract objects; 2) Find all reads/writes that access the shared abstract objects; 3) Detect data races by checking the lockset and happens-before relation in the SHB graph.

To scale SWORD, we leverage a parallel design based on Akka framework [7], which supports efficient message passing and asynchronous communication. With Akka, we can utilize the multi-core on single computer, or deploy the tool on remote clusters. In our implementation, we construct a BugHub that uses BalancingPool to dispatch tasks and manage the status of all BugWorkers. BugWorkers then carry out each step to detect data races.

V. Evaluation

We evaluated the accuracy and performance of SWORD by comparing with RacerD. Since RacerD only performs a partial analysis by default, we ran two sets of tests for it. First, we ran RacerD and SWORD directly on all benchmarks and collected all classes traversed by SWORD. Then we annotated all recorded classes using the @ThreadSafe annotation, and re-ran RacerD on all annotated benchmarks. After annotating, we made sure both tools achieve the exact same code coverage. The testing environment is a 4-core 2.5 GHz Intel x64 Macbook Pro with 16GB RAM. All standard and external libraries are excluded, because RacerD does not consider libraries. The number of BugWorkers in SWORD was set to 8. The benchmarks we use are collected from Dacapo [8] and Petablosx [9], which cover different sizes of real-world Java programs.

Accuracy. The Alarms columns in Table II show the reported race number of SWORD and RacerD. The soundness of SWORD is guaranteed by PTA and SHB graph design. In order to verify the accuracy for all benchmarks, we compared SWORD’s reports with RacerD’s reports on annotated benchmarks, because they had the same code coverage. We manually checked all reported races in the first 4 benchmarks and confirmed all extra alarms reported by RacerD were false positives. The rest 3 large benchmarks all contain more than a thousand alarms, therefore we checked them by sampling. We categorized all reported races by their classes. For each class, we randomly pick three alarms to check:

1) The alarm is reported by RacerD but not by SWORD;
2) The alarm is reported by SWORD but not by RacerD;
3) The alarm is reported by both tools.

After checking all the sample alarms, we conclude that, for the first type of alarms, all of them are false positives. For the second type, there exist some real races, which means RacerD has false negatives. The last type of alarms consists of both false positives and real races.

![Fig. 2: A false positive reported by RacerD in the webblech benchmark, due to not reasoning the thread interleaving.](Image)

![Fig. 3: A false negative reported by RacerD in the sor benchmark, due to the lack of points-to analysis.](Image)

We use two examples to explain the imprecision of RacerD. Figure 2 is a code segment of webblech. Spider is a class implementing Runnable, and assign itself to multiple threads in the constructor. However, the readCheckpoint() will only be called once in main before spider.start(), therefore there is no concurrent access with the queue in this method. Since RacerD does not reason about happens-before relation, readCheckpoint() is reported to be in a race.

Figure 3 is a simplified code piece from sor. Class sor_first_row extends Thread. The Parameters s and e are consecutive ranges (e.g., 1 to 5; 6 to 10). The static arrays black and red in class Sor are assigned to local variables black_ and red_ in sor_first_row. All threads perform array accesses on black_ and red_ therefore manipulating static arrays black and red concurrently. The indexes of these array

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>SWORD Alarms*</th>
<th>Runtime</th>
<th>RacerD Alarms*</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp</td>
<td>454</td>
<td>6/44</td>
<td>1.2s</td>
<td>46</td>
<td>1s</td>
</tr>
<tr>
<td>elevator</td>
<td>1088</td>
<td>1/52</td>
<td>2.8s</td>
<td>45</td>
<td>1s</td>
</tr>
<tr>
<td>webblech</td>
<td>1322</td>
<td>10/53</td>
<td>2s</td>
<td>13</td>
<td>1.4s</td>
</tr>
<tr>
<td>sor</td>
<td>7176</td>
<td>44/64</td>
<td>3.3s</td>
<td>8</td>
<td>1.1s</td>
</tr>
<tr>
<td>sunflow</td>
<td>24713</td>
<td>59/1632</td>
<td>10.3s</td>
<td>1150</td>
<td>135s</td>
</tr>
<tr>
<td>lusearch</td>
<td>48128</td>
<td>222/2786</td>
<td>25s</td>
<td>1420</td>
<td>13s</td>
</tr>
<tr>
<td>avrora</td>
<td>70057</td>
<td>79/4788</td>
<td>30s</td>
<td>27</td>
<td>74s</td>
</tr>
</tbody>
</table>

* # of alarms without annotations / # of alarms with annotations
† RacerD has similar performance with or without annotations

We refer our readers to [1] for a detailed description of the BugHub. For details about the implementation, we use our BugHub to carry out each step to detect data races.
accesses will overlap at $j-1$ and $j+1$. However, RacerD treats black and red as thread local variables due to the missing of precise pointer analysis. As a result, RacerD fail to detect the real races on black on line 16 and red on line 17, no matter it runs with or without annotations.

We can see that without annotation, RacerD can detect only a very limited number of races, and it misses some known real races due to its unsoundness. When fully annotated, RacerD detects many more false positives than SWORO. For benchmarks sor and avrorra, a large portion of their source code cannot be reached during execution, but RacerD still reported many races in unreachable parts. SWORO is more precise than RacerD because it traverses call graphs starting from the program entry points only.

**Performance.** For small benchmarks, both tools manifest a fast detection speed. When the programs grow large, RacerD tends to have a better performance on average, because it utilizes a cheap ownership analysis to replace PTA. This design makes the performance of RacerD mainly depends on program sizes and is not affected by complexity. In contrast, SWORO performs a context-insensitive PTA, which is more precise but relatively more expensive. Nevertheless, SWORO can complete the detection within 3 minutes for all the benchmarks. The performance of SWORO is disproportionate to the program size. For example, sunflow has smaller program size than lusearch, but it contains more abstract threads and shared memory accesses, which require significantly more computation (135s vs 13s) to determine the happens-before relations and locksets.

**Discussion.** In general, both RacerD and SWORO are fast. When targeting at some specific code segments, RacerD is fairly handy, but SWORO is much more precise on whole program scale. Moreover, SWORO provides a comprehensive IDE integration for developers to debug the races. To understand false positives in SWORO, we carefully studied the webblech benchmark, and confirmed that 3 out of the 13 alarms are false positives. The main cause of false positives is context-insensitivity. This can be mitigated by applying a context-sensitive PTA or performing additional analysis based on SWORO to further filter out those false alarms.

**VI. RELATED WORK**

Data races can be detected both statically and dynamically. Dynamic analysis [10]–[13] can detect real bugs in the program, but it is hard to reason about all possible interleaving between threads. Static tools [3], [4], [14] usually provide high code coverage along with many false positives.

Some other techniques [15]–[18] exploit multicore CPU to speed up PTA using parallelism. These techniques can be integrated with SWORO to further improve the performance.

**VII. CONCLUSION AND FUTURE WORK**

We have presented SWORO, a static whole program race detector for Java. We evaluated SWORO with RacerD on various open source projects, and found that SWORO can reveal more real races with fewer false positives and also have a comparable performance in most cases, therefore making SWORO a competitive out-of-box whole program race detector available. In the future, we plan to further consider common data race patterns and leverage techniques like selective context-sensitive PTA or a distributed system design to make SWORO more precise and scalable. The Eclipse JDT language server [19] is a Java language specific implementation of the Language Server Protocol [20]. This also makes it possible to deploy SWORO at remote servers to utilize more powerful computation resources and to work for different programming languages.

**ACKNOWLEDGMENT**

We would like to thank the anonymous reviewers for their valuable feedback on early drafts of this paper. This work was supported by NSF awards CCF-1552935 and CNS-1617985.

**REFERENCES**


