Thread-Sensitive Data Race Detection for Java

Stefan Schulz, Emanuel Herrendorf, Christoph Bockisch
Department of Mathematics and Computer Science
Philipps-Universität Marburg, Germany
{schulzs,bockisch}@mathematik.uni-marburg.de

***Abstract*—In this paper we present StaTS, a precise static data-race detection mechanism for Java. It analyzes applications in four phases. The first one is a novel points-to analysis that includes approximations of threads and execution contexts. The second phase uses the graphs to compute which fields are accessed by which threads, considering the locks held by the threads. The third phase carries out a context-sensitive static happens-before analysis to rule out accesses in execution contexts that can never be executed in parallel. The final phase builds upon the results of the first three to determine conflicting accesses and report them to the user. Our proof-of-concept implementation does not scale for large programs, which is why it can optionally limit the number of points-to relations it considers, based on sampling. Nevertheless, our evaluation shows that—even with sampling enabled for large programs—StaTS detects more data races than existing approaches. In terms of execution time, the analysis without sampling takes in the order of seconds for smaller programs. For larger ones and with sampling, analysis takes minutes, thus being practically usable in nightly build environments in all cases.**

***Index Terms*—Debugging, Fault Prediction, Data Race Detection, Whole Program Analysis, Java.**

I. Introduction

As concurrency is an ever-growing aspect of modern computer programming, handling concurrency bugs efficiently is crucial to allow developers focusing on the implementation of software instead of spending most their time fixing it. Data races [1] are a common type of bug that occur in concurrent applications. They arise when two threads access the same memory location, where at least one of them is writing, without enforcing an order between them. This can lead to non-deterministic behavior, depending on which access happens first. A common approach to detect such bugs is static program analysis [2]–[9]. Application code is traversed to identify potentially conflicting accesses to the global state, i.e. static and object fields. A key aspect of these analyses is constructing so-called points-to graphs [10], [11], which determine the sets of objects that may be referenced by each field or variable. Note that, because the points-to analysis is static, the target of a reference variable has to be overapproximated, i.e, a reference can point to multiple objects at the same time. Two statements are potentially in conflict if they access a field or variable with overlapping sets of objects in the points-to graph. Previous approaches [8], [12]–[16] and studies [17] have shown that introducing context-sensitivity [15], [16] to static analysis tools greatly enhances their precision at the cost of scalability. In addition, some static data race detection tools include a static happens-before analysis [6], [18], [19]. This determines a partial order in which statements are executed and reduces the number of false positives by ruling out potentially conflicting accesses that cannot happen concurrently.

In this paper we present StaTS (**Sta**tic **T**hread-**S**ensitive Data Race Detection), a novel approach to static data race detection. It combines all of the analysis techniques discussed above and introduces the concept of *thread-sensitivity* to them. Thread-sensitivity is an extension of context-sensitivity, where execution contexts are additionally defined in terms of the threads they are executed by. This leads to more fine-grained analyses, because it lets us differentiate between executions of the same method on the same object in different threads. Existing context-sensitive approaches only differentiate between executions on different objects, disregarding the threads.

After presenting the general idea behind StaTS, we formally define the semantics of our thread-sensitive analysis using a top-down approach. We start with the definitions of the *Points-To Relation*, and the set of *Thread Accesses*, each followed by the definition of functions to compute them for a given Java program. Afterwards, we define a static happens-before analysis in terms of a function over thread accesses and establish conditions for detecting conflicts. The introduction of thread-sensitivity allows for two different conflict detection approaches. The first is a precise mode, where parts of the application state are determined in advance, meaning only true data races are reported. The second is a lenient mode, where accesses may be in conflict, depending on the run-time state. This mode may report false data races, but potentially identifies true races that the precise mode misses.

We provide a prototypical implementation of our approach, which we evaluate in terms of precision as well as runtime performance when applied to common benchmark applications. We were able to show that StaTS can achieve perfect precision when using the precise conflict detection mode.

We found that our prototype can analyze smaller programs of up to 10k lines of code in under a second. However, for larger benchmarking applications—such as sunflow and avrora [20]—it did not terminate within a reasonable time. We introduce a sampling-based approach [21] for these applications, where the conflict detection only considers a randomly selected points-to relations per node. Even with sampling enabled, StaTS detected more conflicts than other approaches in most runs. Execution times depend on the sample selection. For large programs with up to 90k lines of code, it ranged from 5 to 20 minutes in most cases, with a maximum of 66 minutes, which is still practical for use in nightly builds.

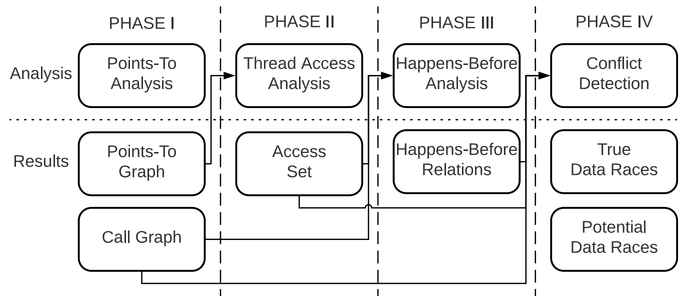

Fig. 1. StaTS data race detection workflow.

II. Static Thread-Sensitive Data Race Detection

Our static thread-sensitive data race detection (StaTS) operates in four phases to identify data races in a given Java application. Figure 1 illustrates the workflow of StaTS.

Firstly, a *Points-To Analysis* is carried out. It generates a points-to graph, which has local variables and *abstract objects* as nodes, and reference values assigned to local variables and fields as edges. Note that only assignments of non-primitive values are considered, as the purpose of the points-to-graphs is to depict how objects are interconnected. Usually abstract objects approximate objects existing at run-time, i.e. one abstract object generally represents a set of run-time objects. We extend the approach proposed by Rountev, Milanova, and Ryder [11] and introduce thread-sensitivity to abstract objects, which leads to a finer grained abstraction. *Thread-Sensitivity* is an extension of object-sensitivity [15], where not only the object a method is invoked on is considered, but also the thread that runs the code. This gives us the ability to distinguish between objects created by the same statement in different method invocations. Consequently, abstract objects in our approach only represent a singular object at run-time. We traverse all reachable statements from a user-selected `main` method, and generate the points-to graph of its execution path iteratively. Since later phases of the analysis need a call graph as input, it is also generated during this first phase.

Secondly, we determine which fields are accessed by the different threads and which locks are held when executing synchronized code. Using the points-to graph as input, the *Thread Access Analysis* calculates all thread accesses in the execution path and returns a set of accesses. We conclude this phase by removing access pairs from the list of potential conflicts, if we can determine that the they occur within the same thread or with common lock objects.

Thirdly, our approach utilizes a novel, thread- and control-flow-sensitive *Static Happens-Before Analysis* [6], [19] to eliminate false positive conflicts. Given a potentially conflicting pair of accesses for the statements s_1 and s_2, we try to predetermine whether one will always be executed before the other at run-time. Note, that the specific order in which they are executed is not important in this context. If an execution order for s_1 and s_2 can be established, the potential conflict is a false positive and can be ruled out.

```
 1   class Main {
 2     public static void main(String[] args) {
 3       new Spider().start();
 4   }}
 5   class Spider {
 6     private DownloadQueue queue;
 7     private long last;
 8     public void start() {
 9       for (int i = 0; i < Config.numThreads; i++) {
10         Thread t = new Thread(this, "T" + (i + 1));
11         t.start();
12     }}
13     public void run() {
14       checkpointIfNeeded()
15       //Download assigned website files
16     }
17     private void checkpointIfNeeded() {
18 ►     if (getTime() − last > Config.interval) {
19         synchronized (queue) {
20           //Create checkpoint
21 ►         last = getTime();
22   }}}}
```

Listing 1. Example data race from WebLech.

Finally, the *Conflict Detection* combines the results of the previous three phases to detect and report conflicting accesses. StaTS offers the use of two modes: Precise and lenient. Precise mode only reports data races where the conflicting accesses are unambiguous, meaning they both access the exact same abstract object and they do not share any common lock objects. Lenient mode reports data races where a program state exists in which two accesses are conflicting. The most notable difference to precise mode is that the accesses can refer to multiple abstract objects, as long as they share at least one common target. The same logic applies to common lock objects. In essence, precise mode reports true positive conflicts by under-approximating race conditions, while lenient mode over-approximates them and may report false positive conflicts.

A. Introduction of Running Example

To illustrate our approach, consider the code example in Listing 1. It shows a simplified version of WebLech URL Spider[1], a Java tool for downloading/mirroring websites. The data race involves the unsynchronized field read on `last` in line 18 and the synchronized field write in line 21. Whenever a thread is started in line 11, it checks whether a new checkpoint should be created. Since the reading access is not synchronized, it is possible that multiple checkpoints are created over a short period of time. This can happen if a thread determines whether a checkpoint should be created, while another is currently creating one and has not updated the `last` field yet. Furthermore, the threads have to wait until the lock object `queue` is available again, before they can start generating the superfluous checkpoints.

When examining this code, our analysis first creates the points-to graph shown in Figure 2. The nodes and edges of the graph showcase one of the main novelties of our analysis. Since they include the execution-context in which the points-to

[1]See http://weblech.sourceforge.net/.

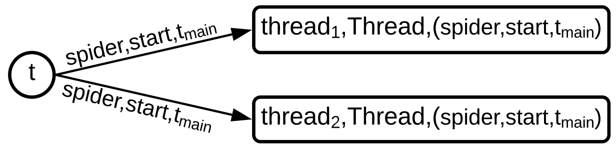
t
spider,start,t_main
spider,start,t_main
thread_1,Thread,(spider,start,t_main)
thread_2,Thread,(spider,start,t_main)

Fig. 2. Points-to graph.

1) Read in t_1 on `spider.last` without lock.
2) Write in t_1 on `spider.last` with lock on `queue`.
3) Read in t_2 on `spider.last` without lock.
4) Write in t_1 on `spider.last` with lock on `queue`.

Fig. 3. Reported accesses.

relations are created, we can later use them to decide whether a conflict can actually happen at run-time.

It cannot be statically determined how many threads are created during the execution of the loop. But assuming two loop executions is sufficient to identify potential conflicts. Hence, we assume that two different threads ($\texttt{thread}_1$, $\texttt{thread}_2$) are assigned as the values of `t`, and add according edges to the graph.[2] The edges mean that the local variable `t` points to $\texttt{thread}_1$ and $\texttt{thread}_2$ in the context of the method `start`, executed on the object instance `spider` by the main thread $\texttt{t}_{main}$. Both `Thread` objects are created in this execution context as well.

Using the points-to graph, the thread accesses are calculated. The result is shown in Figure 3. Since both threads share the same instance of `Spider` and both execute the `checkPointIfNeeded` method, they potentially race for reading/writing access on `last`.

Finally, the static-happens-before analysis lets us conclude that no execution order can be determined for the access pairs 1 and 3, as well as 2 and 4. Both threads are created in the exact same context and therefore are alive at the same time. Since the accesses do not share common locks and use the same instance field from different concurrent threads, they are considered racing and reported as conflicts.

B. Points-To Analysis

We base our analysis on the work presented by Rountev, Milanova, and Ryder [11], and extend it by introducing *thread-sensitive execution contexts*. We define execution contexts as a combination of a method, the abstract object on which the method is executed, as well as the thread executing the method. With this modification, the nodes and edges are thread- and context-sensitive, meaning that two nodes can only be in conflict with each other if they point to the same target node from unrelated contexts in different threads. The goal of this analysis is to generate a points-to graph where the nodes either represent local variables in methods or abstract objects. The edges model points-to relations between the nodes, i.e. from a local variable to an object, or from one object to another via fields. Note that local variables can only be the source of an edge, but never the target. We consider the execution contexts for all edges, as well as the abstract objects. They do not have to be considered for local variables, since the contexts are implicitly modeled by their outgoing edges. The advantage of this is that we do not have to add duplicate nodes to the points-to graph that represent the same local variables in different contexts.

[2]Simulating more loop iterations is redundant, as it would only result in additional conflicts being reported on the same statements, only differing from already reported conflicts in the executing threads.

We formalize the different components for building the points-to graphs. Our definitions are based on the following sets, assuming they are determined prior to the the analysis:

- S (statements)
- V (local variable declarations)
- F (field declarations)
- $M \subseteq V^+ \times (V \cup \epsilon) \times S^+$ (methods)
- $C \subseteq F^+ \times M^+$ (classes)

Methods are defined as tuples of local variables, a return value, and statements. We denote them as

$$m := ((p_1, \ldots, p_i, v_1, \ldots, v_j), ret, (s_1, \ldots, s_n))$$

where the elements $p_i \in V$ represent its parameters, $v_j \in V$ the local variables, and $ret \in (V \cup \epsilon)$ the method's return value, which we treat as an output variable. Since `void` methods do not return values, we use ϵ as placeholders. By doing this, we do not have to use different semantics for `void` and non-`void` methods. The elements s_i represent the implementation of the method.

We now define the sets for the nodes and edges of the points-to graphs. We use the set O to represent abstract objects and CTX for execution contexts. Since we consider execution contexts to determine where and when an abstract object may be created, as well as determining which method was executed on which abstract object by which thread, O and CTX are mutually dependent and thus defined recursively. The initial elements in O and CTX will be discussed later in this section. As mentioned previously, we also consider the context for the edges. Furthermore, we use the term *allocation site* to describe statements in the code that create new objects, i.e. constructor calls via `new` statements.

Definition 1 (Points-To Graph)**.** *Let A be the set of all allocation sites and $T \subseteq O$ the set of started threads in an application. The set of abstract objects O and the set of execution contexts CTX are defined as:*

$$O \subseteq (A \times C \times CTX)$$
$$CTX \subseteq (O \times M \times T)$$

Relations are modeled using the sets E_F, for relations between objects, and E_V, for relations between variables and objects:

$$E_F \subseteq O \times (F \times CTX) \times O$$
$$E_V \subseteq V \times CTX \times O$$

Let $\mathcal{N} := O \cup V$ be the set of nodes and $\mathcal{E} := E_F \cup E_V$ the set of edges. We define the set of points-to graphs G as:

$$G \subseteq \mathcal{P}(\mathcal{N}) \times \mathcal{P}(\mathcal{E})$$

We make additions to these sets to accommodate for essential concepts of the Java programming language:

a) Static Fields and Methods: In order to be able to create points-to relations involving static fields, we add a special abstract object to O for each class. The objects have the form (ϵ, c, ϵ), where $c \in C$ is the class containing the field. They are also used in the execution contexts of static methods.

b) Main Thread: The main thread is not explicitly started in the application itself. We use the abstract object of the `Thread` class to represent it $t_{main} := (\epsilon, Thread, \epsilon)$.

c) Initial Context: We create an initial context to bootstrap our analysis. Since multiple `main` methods may be present, the starting point of the analysis has to be chosen by the user. Assuming the method $m_s \in M$ in class $c_s \in C$ is selected, the initial context is $ctx_{main} := ((\epsilon, c_s, \epsilon), m_s, t_{main})$.

d) Arrays: When an array access occurs, the index that is accessed may be computed dynamically. We add the dummy element $\diamond$ to F and handle array instructions pretending they access the whole array at once.

C. Constructing the Points-To Graphs

We define the $handle$ function to process application code and construct the according points-to graph. Starting from a user-selected `main` method, we traverse the code and modify the graph depending on the encountered statements.

Definition 2 (Points-To Semantics)**.** *The handle function is used to calculate the successor of the points-to graph g with $g := (N, E) \in G$ based on the semantics for statement $s \in S$ and context $ctx \in CTX$. handle is defined as:*

$$\begin{aligned} &handle: \ S \times CTX \times G \rightarrow G \\ &handle(s, ctx, g) = g' \end{aligned}$$

Where $g' := (N', E')$, with $N' \supseteq N$ and $E' \supseteq E$, is the successor graph of g.

The analysis adds new nodes and edges to existing points-to graphs based on the statements encountered. Without the loss of generality, we assume that the analyzed code is in static single assignment form, i.e. interim results and return values are stored to synthetic local variables[3]. Furthermore, assignments are implemented so that the right-hand side is either a value, a constructor call, or a method invocation. The following statements—where $l, r \in V$ are local variables—have to be considered since they affect the points-to graph:

1) Direct assignments: `l = r`
2) Field read: `l = r.f`
3) Field write: `l.f = r`
4) Array read: `l = r[x]`
5) Array write: `l[x] = r`
6) Method invocation: `l = r.m(`$p_1, \ldots, p_k$`)`[4]
7) Object creation: `l = new C(`$p_1, \ldots, p_k$`)`
8) Assignment with cast: `l = (c)r`

[3]Our proof-of-concept implementation simulates these assignments to synthetic variables without altering the code.

[4]For `void` methods assume `l` is a synthetic variable.

In Java, the invoked method depends on the dynamic type of the receiver object. As our abstract objects are characterized by their creation site, their dynamic type is known. The function $lookup(m, o)$ is defined below, whereby m_c is the implementation of method m executed for a receiver of type c according to Java's dispatch semantics.

$$\begin{aligned} &lookup: \ M \times O \rightarrow M \\ &lookup(m, (a, c, ctx)) = m_c \end{aligned}$$

Furthermore, we assume access to the helper methods $class(o)$ and $isA(c_1, c_2)$ to help determining class relations between objects. $class$ returns the compile-time class for a given object, while isA determines how two classes are related:

$$\begin{aligned} &class: \ O \rightarrow C \\ &class(o) = c \end{aligned}$$

$$\begin{aligned} &isA: \ C \times C \rightarrow true, false \\ &isA(c_1, c_2) = \begin{cases} \text{true, if } c_1 = c_2 \\ \text{true, if } c_1 \text{ is a subclass of } c_2 \\ \text{false, otherwise} \end{cases} \end{aligned}$$

In order to process these statements incrementally, we need to be able to resolve the already existing points-to relations for a given node in a given context.

Definition 3 (Points-To Relations)**.** *We use the functions pts_V and pts_F to resolve the points-to relations between the different nodes. pts_V returns the objects a variable points to in a given context, pts_F returns the objects a given field—disregarding the context—refers to:*

$$\begin{aligned} &pts_V: \ \mathcal{P}(\mathcal{E}) \times V \times CTX \rightarrow \mathcal{P}(O) \\ &pts_V(E, v, ctx) = \{o_r \mid (v, ctx, o_r) \in E\} \end{aligned}$$

$$\begin{aligned} &pts_F: \ \mathcal{P}(\mathcal{E}) \times O \times F \rightarrow \mathcal{P}(O) \\ &pts_F(E, o_f, f) = \{o_r \mid \exists ctx \in CTX : (o_f, (f, ctx), o_r) \in E\} \end{aligned}$$

The context can be ignored when determining the points-to relations of fields. Since they belong to an object, they are part of the global state of the program and can be accessed from any context as long as their owner object is alive. However, the context will play a crucial role when determining the order in which field accesses are executed. Next to the pts functions, we need an additional one to calculate the effects of method invocations on the points-to graph:

Definition 4. *Let $handle_s(ctx, g) := handle(s, ctx, g)$ and $m := (p_1, \ldots, p_k, v_1, \ldots, v_i, ret, s_1, \ldots, s_j)$. We define $resolve_p$ to calculate the effects of method invocations:*

$$\begin{aligned} &resolve_p: \ O \times M \times V^+ \times V \times CTX \times G \rightarrow G \\ &resolve_p(o_r, m, r_1, \ldots, r_k, l, (m_c, o_c, t_c), (N, E)) = \\ &\quad handle(l = ret, (m_c, o_c, t_c), handle_{s_j} \circ \cdots \circ handle_{s_1} \\ &\quad \circ\, handle_{p_k = v_k} \circ \cdots \circ handle_{p_1 = v_1}((m, o_r, t_c), (N, E)) \end{aligned}$$

$$handle(l = r, ctx, (N, E)) = \\ (N, E \cup \{(l, ctx, o_r) \mid o_r \in PT_r\}) \quad (1) \\ \text{where } PT_r := pts_V(E, r, ctx)$$

$$handle(l = r.f, ctx, (N, E)) = \\ (N, E \cup \{(l, ctx, o_{rf}) \mid o_r \in PT_r \wedge \exists o_{rf} \in PT_{rf}\}) \quad (2) \\ \text{where } PT_r := pts_V(E, r, ctx);\ PT_{rf} := pts_F(E, o_r, f)$$

$$handle(l.f = r, ctx, (N, E)) = \\ (N, E \cup \{o_l, (f, ctx), o_r) \mid o_l \in PT_l \wedge o_r \in PT_r\}) \quad (3) \\ \text{where } PT_l := pts_V(E, l, ctx);\ PT_r := pts_V(E, r, ctx)$$

$$handle(l = r[x], ctx, (N, E)) = \\ (N, E \cup \{(l, ctx, o_{r\diamond}) \mid o_r \in PT_r \wedge o_{r\diamond} \in PT_\diamond)\}) \quad (4) \\ \text{where } PT_r := pts_V(E, r, ctx);\ PT_\diamond := pts_F(E, o_r, \diamond)$$

$$handle(l[x] = r, ctx, (N, E)) = \\ (N, E \cup \{(o_l, (\diamond, ctx), o_r) \mid o_l \in PT_l \wedge o_r \in PT_r\}) \quad (5) \\ \text{where } PT_l := pts_V(E, l, ctx);\ PT_r := pts_V(E, r, ctx)$$

$$handle(l = r.m(r_1, \ldots, r_k), ctx, (N, E)) = \\ \bigcup_{o_r \in PT_r} resolve_p(o_r, m_r, r_1, \ldots, r_k, l, ctx, (N, E)) \quad (6) \\ \text{where } PT_r := pts_V(E, r, ctx); m_r = lookup(o_r, m)$$

$$handle(l = new\ c_i(r_1, \ldots, r_k), ctx, (N, E)) = \\ resolve_p(o_c, init_c, r_1, \ldots, r_k, l, ctx, (N \cup o_c, E)) \quad (7) \\ \text{where } o_c := (a_i, c, ctx)$$

$$handle(l = (c)r, ctx, (N, E)) = \\ (N, E \cup \{(l, ctx, o_r) \mid o_r \in PT_r \wedge isA(c_{o_r}, c)\}) \quad (8) \\ \text{where } PT_r := pts_V(E, r, ctx); c_{o_r} := class(o_r)$$

Fig. 4. Semantics of the *handle* function.

When resolving a method call, the arguments $r_1, \ldots, r_k$ are assigned to the method's parameters $p_1, \ldots, p_k$ and each statement of the method is handled. Afterwards, the invocation is concluded by assigning the return value to l. For simplicity, we assume the return values of constructors are the objects they create. The statements of the resolved method are handled in a different context, since they are executed in m, not m_c.

Using $resolve_p$ and the pts functions, we define the semantics for $handle$ based on the set of Java statements mentioned earlier in this section[5]. The full semantic definition is shown in Fig. 4. In definition (7) a_i is the allocation site associated with the `new` operator for c_i, $init_c$ is the according constructor.

Most semantic definitions only add new edges to the points-to graph. The general idea is to resolve the existing points-to relations of the right-hand side of the expression and add them to the local variable or object on the left-hand side. A special case occurs when a class cast is used before an assignment statement (8). We leverage typing relationships to only transfer points-to relations from the right-hand to the left-hand side if the cast is valid for the referenced abstract object's type.

[5]For two graphs $g_1 := (N_1, E_1), g_2 := (N_2, E_2)$ we define the union operator so that $g_1 \cup g_2 := (N_1 \cup N_2, E_1 \cup E_2)$

Semantic definitions (6) and (7) handle method invocations. In essence, they simulate a method dispatch and recursively use the $handle$ function to iterate over the statements of the invoked method. For object creations (7), the created objects are added to the points-to graph as nodes.

Static field accesses and method invocations are treated analogously to their virtual counterparts, using the abstract objects of their classes instead of instances. Since the number of loop iterations cannot be statically determined, we assume loops are executed twice, in order to detect conflicts between loop iterations. This means two abstract objects are generated per allocation site and statements in loops are handled twice.

D. Thread Access Analysis

We use the points-to graph to carry out the thread access analysis. The goal is to determine whether two events executed in different threads might affect each other. To calculate thread accesses, the following statements have to be considered, since they interact with the global state of an application:

1) Field read: `l = r.f`
2) Field write: `l.f = r`
3) Array read: `l = r[x]`
4) Array write: `l[x] = r`
5) Method invocation: `l = r.m(`$p_1, \ldots, p_k$`)`
6) Synchronized blocks: `synchronized(r)`$\{s_1, \ldots, s_n\}$
7) Object creation: `l = new C(`$p_1, \ldots, p_k$`)`

A thread access consists of a set of abstract objects, a field, the access type (read/write), the objects the access might hold as a lock, and the executing thread. Locks are handled as sets of objects and consequently the set of all locks is a set of sets of objects. The distinction between lock sets will become relevant for the conflict detection in the last phase of the analysis. As mentioned earlier, local variables can refer to multiple objects, thus we have to consider all of them when a relevant statement is encountered.

Definition 5 (Thread Access)**.** *Let $\mathcal{L} \subseteq \mathcal{P}(O)$ be the set of lock objects. We define $\mathcal{X}$, the set of thread accesses as:*

$$\mathcal{X} \subseteq \mathcal{P}(O) \times F \times \{read, write\} \times \mathcal{P}(\mathcal{L}) \times T$$

A thread access $x = (objects, field, type, locks, thread)$ is a tuple consisting of a set of objects, the field that is accessed, the type of the access, a set of possible locks—where a lock is a set of objects—and the thread that executes the access.

Definition 6 (Access Semantics)**.** *We define the access function that returns a set of thread accesses $X \in \mathcal{P}(\mathcal{X})$ for a statement $s \in S$, the context $ctx \in CTX$ it is executed in, a points-to graph $(N, E) \in G$ and a set of active locks $L \in \mathcal{P}(\mathcal{L})$:*

$$access: \ S \times CTX \times G \times \mathcal{P}(\mathcal{L}) \to \mathcal{P}(\mathcal{X}) \\ access(s, ctx, (N, E), L) = X$$

Again, we need an additional function to process method invocations. In contrast to $resolve_p$ we do not have to consider the bindings of the method's arguments and its return value as they are already accounted for in the points-to graph.

Definition 7. *Let $m := (\ldots, s_1, \ldots, s_n)$ be a method that is invoked in the analyzed code and needs to be resolved for further analysis. We define $resolve_x$ to calculate the field accesses in invocations in m:*

$$resolve_x : \; O \times M \times CTX \times G \times \mathcal{P}(L) \rightarrow \mathcal{P}(X)$$
$$resolve_x(o_i, m, (o_c, m_c, t_c), (N, E), L) =$$
$$\bigcup_{i \in \{1,\ldots,n\}} access(s_i, (o_i, m, t_c), (N, E), L)$$

Similar to the $resolve_p$ function, $resolve_x$ simulates a context switch by changing the abstract object and method in the execution context for the statements in m.

With the use of $resolve_x$, we define the semantics of $access$ for method invocations, field and array accesses. We assume access to the helper function $isSync(m)$, which returns whether a method is synchronized. Figure 5 shows the full semantic definition for all statements that are relevant for the thread access analysis. We assume $ctx := (o_c, m_c, t_c)$ is current execution context.

Most of the semantic definitions simply return a singular access tuple that contains all objects the accessed field may belong to. For method invocations (5), synchronized blocks (6), and constructor calls (7), the access sets for all their statements are returned. If an invoked method is synchronized, the receiver object is considered locked for all statements in the method. For synchronized blocks, all objects referenced by the variable have to be considered locked.

$$access(l = r.f, (o_c, m_c, t_c), (N, E), L) = \{(PT_r, f, read, L, t_c)\} \quad (1)$$
where $PT_r := pts_V(E, r, ctx)$

$$access(l.f = r, (o_c, m_c, t_c), (N, E), L) = \{(PT_l, f, write, L, t_c)\} \quad (2)$$
where $PT_l := pts_V(E, l, ctx)$

$$access(l = r[x], (o_c, m_c, t_c), (N, E), L) = \{(PT_r, \diamond, read, L, t_c)\} \quad (3)$$
where $PT_r := pts_V(E, r, (o_c, m_c, t_c))$

$$access(l[x] = r, (o_c, m_c, t_c), (N, E), L) = \{(PT_l, \diamond, write, L, t_c)\} \quad (4)$$
where $PT_l := pts_V(E, l, (o_c, m_c, t_c))$

$$access(l = r.m(p_1, \ldots, p_k), ctx, (N, E), L) = \bigcup_{o_r \in PT_r} \begin{cases} resolve_x(o_r, m_r, ctx, (N, E), L \cup \{\{o_r\}\}), & \text{if } m_r \text{ is synchronized} \\ resolve_x(o_r, m_r, ctx, (N, E), L), & \text{otherwise} \end{cases}$$
where $PT_r := pts_V(E, r, ctx)$; $m_r := lookup(m, o_r)$ (5)

$$access(synchronized(r)\{s_1, \ldots, s_n\}, ctx, (N, E), L) = \bigcup_{i \in \{1,\ldots,n\}} access(s_i, ctx, (N, E), L \cup PT_r)$$
where $PT_r\{pts_V(E, r, ctx)\}$ (6)

$$access(l = new\ c(p_1, \ldots, p_k), ctx, G, L) = resolve_x(o_n, init_c, ctx, G, L) \quad (7)$$
where $o_n := (a_i, c, ctx)$

Fig. 5. Semantics of the $access$ function.

E. Static Happens-Before Analysis

Using the call graph and the access sets, we can try to determine the execution order between potentially conflicting accesses using a static happens-before analysis. In this analysis, the exact order in which the accesses are executed is not important, it thus can be simplified.

Definition 8 (Static Happens-Before Relation). *We define shb as the function that returns whether the execution order of two accesses $x_1, x_2 \in \mathcal{X}$ can be determined. Let $x_1 := (s_1, (o, m_1, t_1), g_1, l_1)$, and $x_2 := (s_2, (o, m_2, t_2), g_2, l_2)$. Assuming s_1 and s_2 are field accesses on the same static or instance field:*

$$shb : \; \mathcal{X} \times \mathcal{X} \rightarrow \{true, false\}$$
$$shb(x_1, x_2) = \begin{cases} \text{true, if } t_1 = t_2 \\ \text{true, if } t_1 \text{ starts } t_2 \text{ and} \\ \qquad s_1 \text{ is executed before the start} \\ \text{true, if } t_2 \text{ starts } t_1 \text{ and} \\ \qquad s_2 \text{ is executed before the start} \\ \text{false, otherwise} \end{cases}$$

In the trivial case the statements of both accesses belong to the same scope and are executed by the same thread. Their happens-before relation is determined by their bytecode order.

The more complex case occurs when the accesses are executed by different threads. In this case the call graph needs to be searched for the start events of their respective threads, as well as the execution contexts, and the statements themselves. We can establish the execution order of an access pair if they are either executed by the same thread, or one thread starts the other and one of the accesses is executed before said start.

The execution order of statements across multiple loop iterations is not always clear. For example, the instructions of a nested `if` statement cannot be ordered in relation to the other statements, since they might not be executed in all iterations. Consequently, we cannot determine happens-before relations for them.

F. Analyzing the Running Example

To revisit the example introduced in Listing 1 at the beginning of this Section, we showcase how the simplified WebLech code snippet can be analyzed using StaTS.

The user selects the `main` method and the points-to graph G_a shown in Figure 2 is generated. This is done by applying the handle function to all statements in the method:

$$G_a = handle_{t_2 = t_1.start()} \circ handle_{t_1 = new\ Spider()}(ctx_s, (N_0, \emptyset))$$

N_0 is the initial set of nodes for the application, containing the abstract objects for each class, as well as the local variables in all methods. Note that t_1 and t_2 are synthetic variables used to store the interim results of the statements. Java uses an operand stack to hold interim results of expressions. By introducing synthetic variables we do not have to simulate the stack during analysis, resulting in a more streamlined, simplified approach.

Afterwards the *access* function can be used to determine the thread accesses presented in Figure 3:

$$\begin{aligned}X_a =& access_{t_2=t_1.start()}\\ &\circ access_{t_1=new\ Spider()}(ctx_s, G_a, \emptyset)\\ =&\{a_1 : (spider, Spider.last, read, \emptyset, thread_1),\\ &a_2 : (spider, Spider.last, write, \{queue\}, thread_1),\\ &a_3 : (spider, Spider.last, read, \emptyset, thread_2),\\ &a_4 : (spider, Spider.last, write, \{queue\}, thread_2)\}\end{aligned}$$

Using the call graph and thread accesses, the static happens-before analysis checks the accesses that target the same field by pairs to try determining an execution order between them:

$$\begin{aligned}shb(a_1, a_2) &= true\\ shb(a_1, a_3) &= false\\ shb(a_2, a_3) &= false\\ shb(a_2, a_4) &= false\\ shb(a_3, a_4) &= true\end{aligned}$$

G. Conflict Detection Mechanism

We can now use our analysis results to determine whether these accesses are in conflict with each other. Two accesses are in conflict, if all of the five following conditions apply:

1) Both target the same field.
2) At least one of them is writing.
3) They are executed by different threads.
4) They are not protected by a common lock object.
5) Their execution order cannot be determined.

Conditions 1 to 4 are determined by the points-to and access analyses, condition 5 using the happens-before relations.

Since a variable or field can represent multiple abstract objects, the precision of the conflict analysis may vary depending on how overloaded they are. A conflict might be reported even though it will never occur during execution. To account for this, our analysis supports two modes: Precise and lenient. Both of them are based on the conditions presented above, however they implement conditions 1 and 4 in different ways.

Definition 9.1 (Precise Conflict Detection). *Let $x_1, x_2 \in \mathcal{X}$ be two accesses, with $x_1 \neq x_2$, $x_1 := (s_1, obj_1, f_1, type_1, L_1, t_1)$ and $x_2 := (s_2, obj_2, f_2, type_2, L_2, t_2)$. x_1 and x_2 are in conflict if the following predicates are fulfilled:*

1) $|obj_1| = 1 \wedge |obj_2| = 1 \wedge obj_1 = obj_2 \wedge f_1 = f_2$
2) $t_1 \neq t_2$
3) $type_1 = write \vee type_2 = write$
4) $\forall l_1 \in L_1, l_2 \in L_2 :\ l_1 \cap l_2 = \emptyset$
5) $shb(x_1, x_2) = false$

```
Spider.last
  -Access Conflict 1
     thread1
       [Spider] private void checkpointIfNeeded()
         if (getTime() - last > Config.interval)
     thread2
       [Spider] private void checkpointIfNeeded()
         last = getTime();
  -Access Conflict 2
     thread1
       [Spider] private void checkpointIfNeeded()
         last = getTime();
     thread2
       [Spider] private void checkpointIfNeeded()
         if (getTime() - last > Config.interval)
```

Fig. 6. Reported data race on `Spider.last`.

Definition 9.2 (Lenient Conflict Detection). *x_1 and x_2 may be in conflict if the following predicates are fulfilled:*

1) $obj_1 \cap obj_2 \neq \emptyset \wedge f_1 = f_2$
2) $t_1 \neq t_2$
3) $type_1 = write \vee type_2 = write$
4) $\exists l_1 \in L_1, l_2 \in L_2 : l_1 \cap l_2 = \emptyset$
5) $shb(x_1, x_2) = false$

Conflicts in precise mode are definitive, because both accesses happen on the exact same static or instance field. Because there are no common locks they might be executed concurrently. This mode is designed to achieve *increased precision*, at the risk of *decreased recall*.

Lenient mode detects at least the same conflicts as precise mode. Additional alerts indicate that a scenario exists where the statements involved in a conflict access the same static or instance field without a common lock. This can lead to an *increased recall* at the risk of *decreased precision*.

In our example, both modes determine that the accesses a_1 and a_4, as well as a_2 and a_3 are racing, because these pairs fulfill all five data race conditions. Note that, while the exection order of accesses a_2 and a_4 cannot be established, both of them are reading accesses. They do not affect each other and thus are not reported as conflicting. The reports of both modes are identical, because the points-to graph G_a is unambiguous. The detected racing pairs access the same field, so they are summarized and reported as one data race on the field `Spider.last`. Figure 6 shows the report StaTS presents to the user.

III. Evaluation

We implemented an Eclipse plugin as a proof-of-concept. Our main goal is to evaluate the precision[6] of StaTS compared to RacerD [2]. Note that for static analysis a true positive indicates that there exists a thread schedule in which the reported data race occurs. However, this schedule does not necessarily happen in each execution. There are two important differences in the way StaTS and RacerD analyze applications and report conflicts:

[6]Recall cannot be measured, as the total number of data races is unknown.

TABLE I
BENCHMARK APPLICATION CODE METRICS.

Suite	Benchmark	LOC	#Classes	#Interfaces	#Methods
Petablox	elevator	334	5	0	23
	sor	7176	112	12	936
	tsp	446	4	0	17
	weblech	1309	11	1	92
DaCapo	avrora	91025	1787	83	9011
	sunflow	24721	260	22	1814

TABLE II
STATS RESULTS FOR THE PETABLOX BENCHMARKS.

	Precise Mode		Lenient Mode			
Benchmark	Alerts	Fields	Alerts	Fields	Precision	Time
elevator	0	0	8	4	- / 0%	0.4s
sor	10	2	10	2	100%	1.9s
tsp	4	3	12	9	100%	0.4s
weblech	5	4	5	4	100%	0.9s

TABLE III
RACERD RESULTS FOR THE PETABLOX BENCHMARKS.

Benchmark	Alerts	Fields	Matching Alerts	Precision	Time
elevator	0	0	0	-	0.1s
sor	0	0	0	-	0.3s
tsp	4	2	3	75%	0.1s
weblech	8	4	6	75%	0.1s

- StaTS starts the analysis at a given `main` method and considers all statements that are reachable from it, i.e. it is control-flow- and context-sensitive. In contrast, RacerD is compositional, meaning that "*the analysis result of a composite program can be obtained from the results of its parts*" [2]. Users explicitly state the classes to inspect, which are then analyzed without considering the context they are used in. Context-sensitive program analysis leads to more precise results at the cost of scalability, while compositional analysis leads to better scalability at the cost of precision. In order to make the results of both tools more comparable, we take note of all classes that are traversed by StaTS and pass them to RacerD for analysis.
- StaTS reports conflicting accesses from two different contexts. This includes at least one writing access and multiple other accesses racing with this write. Contrarily, RacerD reports problematic statements either because of unprotected writes or read/write races, i.e. only up to two accesses are reported as conflicting at once. This means that multiple alerts reported by RacerD can correspond to a single alert reported by StaTS and vice versa.

We also tried to gather results for SWORD [6], however it is no longer maintained and we were unable to reproduce the findings presented by the authors. Therefore no comparison between StaTS and SWORD can be provided. We analyze a selection of applications from the DaCapo [20] and Petablox[7] benchmarks. Their relevant metrics are shown in Table I. We calculated these metrics using SourceMeter [22]. All measurements are performed on an AMD Ryzen 7 1700 @ 3.0 GHz system with 32 GB RAM running Ubuntu 20.04.

A. Petablox Benchmarks

Tables II and III show the results for the Petablox benchmarks. The times are the average execution times of five runs. The 'matching alerts' column in Table III shows the number of alerts reported by RacerD that were also reported as conflicting by StaTS.

The soundness of conflicts reported in precise mode is guaranteed because of the unambiguous points-to graph, i.e. parts of the application's run-time state can be predicted. The alerts reported in this mode are all true positives, resulting in a precision of 100%. We manually analyzed the conflicts only reported by RacerD and found that they are false positives. The execution times of StaTS and RacerD are comparable, making StaTS viable for live analysis of smaller applications. The conflicts reported by the different modes of StaTS exactly match for sor and weblech. For elevator, the additional alerts in lenient mode are all *false positives*. The additional alerts reported for tsp are all *true positives*. As mentioned earlier, lenient mode constitutes a trade-off between precision and recall. There was no significant execution time difference between both modes.

B. DaCapo Benchmarks

Scalability issues arise for larger benchmarks, such as avrora and sunflow. This is because for multiple points-to relations with the same source, an alternative execution path has to be considered for each relation when a method is invoked on the referenced objects. This results in an exponential increase of the search space during the graph construction. Because of this, we introduce a sampling-based strategy to our prototype, where only a defined number of points-to edges—chosen randomly—is traversed for each object. We found that a limit of 3 sampled edges achieves the best results at reasonable execution times. For example, increasing the sampling size to 4 raised the analysis time of sunflow to 4 hours, however it did not report additional conflicts.

While random traversal leads to varying results, we observed that this still results in more reported conflicts for most runs compared to RacerD. Tables IV and V show the results of 10 successful analysis runs for avrora and sunflow[8].

RacerD reported 71 alerts on 24 fields, at an average execution time of 2.7s for avrora. For sunflow it reported 45 alerts on 29 fields, at an average of 1.6s. However, RacerD and StaTS did not report any common data races.

[7]See https://github.com/petablox/petablox-bench.

[8]Because of the random selection, some runs reported no conflicts. We repeated those.

TABLE IV
StaTS results for avrora.

Precise Mode		Lenient Mode		
Alerts	Fields	Alerts	Fields	Time
198	115	234	120	1517.6s
83	34	110	39	97.7s
8	8	8	8	12.1s
151	89	188	97	1121.5s
114	62	134	67	147.8s
118	48	153	53	354.7s
194	110	235	118	366.3s
156	86	187	91	1061.9s
174	102	214	110	1514.5s
8	8	8	8	16.1s

TABLE V
StaTS results for sunflow.

Precise Mode		Lenient Mode		
Alerts	Fields	Alerts	Fields	Time
73	9	165	15	3955.2s
29	6	123	9	326.7s
85	13	191	17	1464.1s
21	6	27	6	557.9s
29	5	151	11	333.9s
90	8	184	15	2506.5s
29	6	156	9	581.6s
18	6	24	33	45.6s
34	18	126	38	1260.6s
24	8	39	8	68.8s

Execution times for StaTS vary depending on the sampled points-to relations and range from 12 seconds to 66 minutes. This means that for large-scale projects StaTS is too slow to be used in an IDE, but fast enough to be used as part of a nightly build cycle. We randomly checked some of the reported alerts in both modes and found that all alerts reported in precise mode are true positives. The alerts only reported in lenient mode showed comparable amounts of true and false positives.

Furthermore, the different runs seem to form clusters with similar numbers of reported conflicts. This suggests that there are hotspot nodes in the points-to graph that heavily influence the results of the analysis. If it is possible to determine these nodes, future research can yield an opportunity to achieve more consistent results, for example by dynamically increasing the sampling rate when encountering a hotspot.

In combination with the better performance of RacerD, our results suggest that both approaches are complementary. RacerD provides instantaneous—albeit less precise—data race detection for live analysis in an IDE, while StaTS offers a precise race detection best suited for analyzing nightly builds.

IV. Related Work

The different phases of StaTS heavily rely on two concepts, which typically have to be considered when designing a static code analysis approach. The first concept is context-sensitivity[9] [8], [23]–[27] which is usually used as a means to improve the precision of points-to analyses at the cost of scalability. More recent approaches, such as D4 [7], RacerD [2], and SWORD [6] forgo this feature in order to achieve better scalability when examining large applications. While D4 and SWORD still use a points-to analysis, albeit context-insensitive, RacerD omits this technique altogether in favor of a much cheaper ownership analysis. In contrast to a traditional points-to analysis, where we try to find accesses that might access the same memory address, ownership analysis tries to identify accesses that are clearly safe by checking if the access to a memory address is exclusive to a statement.

The second concept is control-flow-sensitivity [24]–[26]. It is typically used to statically resolve function pointers [25] or lock aquisitions [26] when analyzing applications written in C, resulting in fewer false positive alerts. Again, it trades precision for scalability [8], and to our knowledge there are no other static data race detection approaches for Java that utilize this. However, it is a commonly used in the context of model-checking-based approaches [28], [29].

As shown in our evaluation, precise data race detection significantly impacts performance. There are two well combinable approaches to compensating this drawback. The first is reducing the search space that the analysis has to handle, for example by only considering specific parts of an application, such as methods [2], or analyzing the applications in a layered fashion [7], [19], [23]. Alternatively change-awareness can be introduced to the analysis, so only code changes are considered after the first run [19]. Another solution for reducing the search space—and the one we chose for our approach—is to omit statements from the points-to graph that are not reachable from a selected fix point, such as a `main` method [8], [30].

The second approach is to incorporate developer feedback into the analysis. This is often done by using annotations [2], [11], such as `@ThreadSafe` or `@GuardedBy`. While this requires additional work by developers, it helps guide the analysis and significantly improves performance.

Similar to StaTS, some approaches consider thread interleaving to improve precision. This can be done via static happens-before (SHB) [6], [7], [18], [19] or may-happen-in-parallel (MHP) analysis [8], [31]–[33]. The main difference is that SHB usually aims to eliminate possible conflict candidates, while MHP aims to identify them.

For further work on fault localization we refer to the extensive survey by Wong, Gao, Li, Abrea and Wotawa [34].

V. Conclusion and Future Work

In this work we presented a novel static data race detection approach for Java. In contrast to other approaches, StaTS takes threads and execution contexts into consideration for the construction of points-to graphs. This allows us to introduce an additional thread access detection phase to the analysis and make our SHB analysis thread-sensitive, thus achieving a high precision. However, scalability issues arise for larger

[9]Also known as object-sensitivity [15] for object-oriented languages.

applications, forcing us to fall back to sampling a set amount of points-to edges while traversing the points-to graph. However, even with sampling activated, StaTS detected more race conditions than other approaches in most runs.

In future work, we plan to investigate different options to increase the performance of StaTS. A possible approach is to split the points-to analysis into two stages, considering threads only after constructing an initial points-to graph. This could be used in unison with annotations provided by developers to help further reducing the search space of the analysis.

Additionally, we would like to perform a more in-depth evaluation of larger applications. A key interest is whether it is possible to identify sampling-hotspots, where increasing the sample rate would increase the recall of StaTS with as little impact to the performance as possible.

Acknowledgment

We would like to thank our colleague Jens Kosiol for proofreading this work and providing useful suggestions on how to improve it.

References

[1] C. Artho, K. Havelund, and A. Biere, "High-level data races," *Software Testing, Verification and Reliability*, vol. 13, no. 4, pp. 207–227, 2003.

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

[3] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2002, pp. 211–230.

[4] J.-D. Choi, A. Loginov, and V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," Technical Report RC22146, IBM Research, Tech. Rep., 2001.

[5] C. Flanagan and S. N. Freund, "Type-based race detection for java," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 219–232.

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

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

[8] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 308–319.

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

[10] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[11] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for java using annotated constraints," *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 43–55, 2001.

[12] R. Atkey and D. Sannella, "Threadsafe: Static analysis for java concurrency," *Electronic Communications of the EASST*, vol. 72, 2015.

[13] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 423–434, 2013.

[14] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, pp. 1–53, 2008.

[15] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

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

[17] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *International Conference on Compiler Construction*. Springer, 2006, pp. 47–64.

[18] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, "Precise static happens-before analysis for detecting uaf order violations in android," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 276–287.

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

[20] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[21] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: Proportional detection of data races," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 255–268, 2010.

[22] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "Source meter sonar qube plug-in," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 77–82.

[23] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 307–317.

[24] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.

[25] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering*, 2009, pp. 13–22.

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

[27] 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.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated software engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[29] R. Nakade, E. Mercer, P. Aldous, K. Storey, B. Ogles, J. Hooker, S. J. Powell, and J. McCarthy, "Model-checking task-parallel programs for data-race," *Innovations in Systems and Software Engineering*, vol. 15, no. 3, pp. 289–306, 2019.

[30] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 387–400, 2006.

[31] R. Barik, "Efficient computation of may-happen-in-parallel information for concurrent java programs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2005, pp. 152–169.

[32] L. Li and C. Verbrugge, "A practical mhp information analysis for concurrent java programs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2004, pp. 194–208.

[33] G. Naumovich, G. S. Avrunin, and L. A. Clarke, "An efficient algorithm for computing mhp information for concurrent java programs," in *Software Engineering—ESEC/FSE'99*. Springer, 1999, pp. 338–354.

[34] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.