Effective Techniques for Static Race Detection in Java Parallel Loops

Cosmin Radoi, University of Illinois
Danny Dig, Oregon State University

Despite significant progress in recent years, the important problem of static race detection remains open. Previous techniques took a *general* approach and looked for races by analyzing the effects induced by low-level concurrency constructs (e.g., java.lang.Thread). But constructs and libraries for expressing parallelism at a higher level (e.g., fork-join, futures, parallel loops) are becoming available in all major programming languages. We claim that specializing an analysis to take advantage of the extra semantic information provided by the use of these constructs and libraries improves precision and scalability.

We present ITERACE, a set of techniques that are *specialized* to use the intrinsic thread, safety, and data-flow structure of collections and of the new loop-parallelism mechanism introduced in Java 8. Our evaluation shows that ITERACE is fast and precise enough to be practical. It scales to programs of hundreds of thousands of lines of code and it reports very few race warnings, thus avoiding a common pitfall of static analyses. In five out of the seven case studies ITERACE reported no false warnings. Also, it revealed six bugs in real-world applications. We reported four of them, one had already been fixed, and three were new and the developers confirmed and fixed them.

Furthermore, we evaluate the effect of each specialization technique on the running time and precision of the analysis. For each application, we run the analysis under 32 different configurations. This allows us to analyze each technique's effect both alone and in all possible combinations with other techniques.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification — Reliability, Validation; D.2.5 [**Software Engineering**]: Testing and Debugging — Debugging aids; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages — Program analysis

General Terms: Reliability, Verification

Additional Key Words and Phrases: Static race detection, Java, synchronization, static analysis

1. INTRODUCTION

The recent prevalence of multi-core processors has increased the use of shared-memory parallel programming [Torres et al. 2011]. Most major programming languages have parallel constructs or libraries that provide extensive support for loop parallelism, e.g., Parallel.For in .NET TPL [TPL 2015], .parallelStream() in Java 8 collections [lam 2015], parallel_for in C++ TBB [TBB 2015]. Recent empirical studies [Okur and Dig 2012; Torres et al. 2011]

show that developers have started to adopt high-level concurrency structures, despite their recent introduction.

Still, programs with parallel loops are subject to the plague of shared-memory concurrent programming: data races. A data race can occur when one thread executing a loop iteration writes a memory location and another thread executing another loop iteration accesses the same memory location with no ordering constraint between the two accesses.

The high-level nature of parallel loops does not protect the developer from introducing data races. At most, the high-level structures give the developer a clearer view of the threads that execute concurrently. Still, we have no reason to believe that code using high-level concurrency structures, such as parallel loops, is less susceptible to data races than code using threads directly. As the high-level concurrency structures are easier to use, more developers (including inexperienced ones) may be encouraged to use concurrency. We know that developers often misuse the new high-level concurrency APIs [Okur and Dig 2012; Lin and Dig 2013]. Data races, which are subtler mistakes, should be at least as prevalent, especially in Java, a language where mutability is encouraged.

Data races are hard to find due to non-deterministic thread scheduling. This has led to a large body of research on race detection. Static race detection techniques [Henzinger et al. 2004; Abadi et al. 2006; Naik et al. 2006; Voung et al. 2007; Jhala and Majumdar 2007; Naik and Aiken 2007; Halpert et al. 2007; Bodden and Havelund 2008; Kahlon et al. 2009; Naik et al. 2010; Liang et al. 2010; Pratikakis et al. 2011] use an underlying static model of the program's real execution. In theory, this allows a single analysis pass to find all the races that could occur in all possible program executions. Well-implemented, conservative static race detectors do not miss races but are faced with the opposite problem: despite continuous improvements, they still report too many false warnings. For example, we applied JChord [Naik et al. 2006], a state-of-the-art static race detector, on compute-intensive loops from seven Java applications. In many cases, JChord reported thousands of racing accesses per analyzed loop. This may be one of the reasons why static race detectors have not been embraced in practice. Indeed, most of the recent work on data-race detection has focused on dynamic detectors [Schonberg 1989; Dinning and Schonberg 1990; Choi et al. 1991; Mellor-Crummey 1991; Adve et al. 1991; Savage et al. 1997; Ronsse and De Bosschere 1999; Christiaens and De Bosschere 2001; von Praun and Gross 2001; O'Callahan and Choi 2003; Nishiyama 2004; Marino et al. 2009; Flanagan and Freund 2009; Naik et al. 2010; Liang et al. 2010; Sheng et al. 2011], which typically have much fewer false warnings, but have high overhead and miss races on program paths that are not executed.

Can static race detection for Java applications be practical? Previous approaches embraced *generality*: they tried to work equally well for any kind of parallel construct by analyzing thread-level concurrency, did not differentiate between application and library code, and did not use the documented behavior of libraries. This came at the expense of *practicality*: they were either not scalable or reported a high number of false warnings. We hypothesize that a *specialized* analysis can significantly improve precision while maintaing scalability. In this paper, we validate this hypothesis for the case of Java parallel loops.

Our goals are to prune false warnings and reduce as much as possible the total number of warnings the programmer has to inspect, while not sacrificing safety, i.e., not removing any true race. We present three *specialization* techniques that contribute to these goals:

(1) *2-Threads* – make the analysis aware of the threading and data-flow structure of loop-parallel operations.
(2) *Filtering* – filter the race warnings based on a thread-safety model of library classes.
(3) *Bubble-up* – report races in application code, not in libraries.

We implemented these techniques in a tool called ITERACE. The current implementation only finds data races between the threads of parallel loops, but it could be extended to or
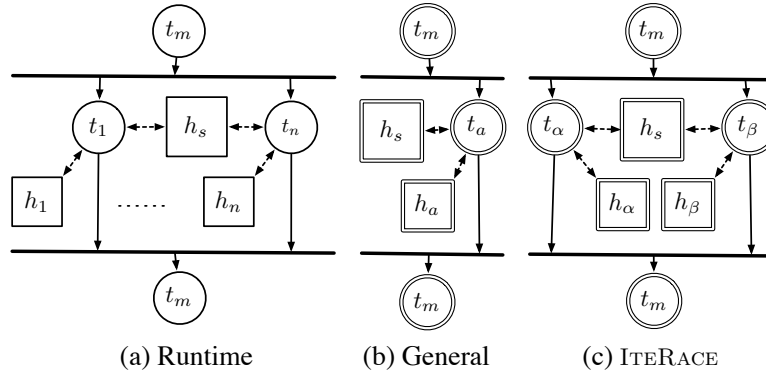
Fig. 1: **Modeling a parallel loop.** Circles are threads, squares heap regions. Double line denotes abstraction.

integrated with a general race detector. We empirically validated how well the proposed techniques work individually, and in various combinations.

1.1. 2-Threads

A parallel loop is an SPMD-style (Single Program, Multiple Data) computation. Its iterations are identical tasks processing different input elements. The tasks are executed by a pool of threads. Without loss of generality, we can consider that each task/iteration is computed by a different thread. The main thread forks multiple identical threads at the beginning of the loop and waits for these threads to join at the end of the loop (Fig.1.a). Each of the threads/iterations can access a part of the heap. In the figure, h_s is the set of objects shared between parallel threads. h_i is the set of objects specific to thread t_i, i.e., input or new objects only accessed by thread t_i.

A general race detector models the identical forked threads by only one abstract thread [Naik et al. 2006; Pratikakis et al. 2011] (see Fig.1.b). This makes the thread-specific object sets $h_1...h_n$ indistinguishable from each other, as they are modeled by a unique set h_a. Then, escape analysis or other techniques are used to refine the results and reduce the number of false warnings.

In contrast, our specialized technique models the identical forked threads by two distinct abstract threads, t_α and t_β (Fig. 1.c). This closely matches the definition of a data race as it disambiguates the two threads involved in the definition. As the objects specific to each of the two threads are modeled by the separate sets h_α and h_β, the number of abstract objects that are shared is significantly reduced. Our modeling subsumes the effect of thread escape analysis but is more precise. Like with thread-escape, an abstract object that does not escape a thread is considered safe. However, when an object does escape, our analysis does not implicitly consider it unsafe. ITERACE only reports a race warning when an object reaches the other abstract thread and there is a concurrent access.

1.2. Bubble-up

All Java programs of real value are built on top of libraries - even the "Hello World" program uses several JDK classes. General race detectors do not keep track of whether the race appears in library code or in application code. However, reporting a race in library code has little practical value for application developers as such a race is rarely due to a buggy library - it is likely due to concurrent misuse of the library.

ITERACE *bubbles-up* the race warnings that occur in library code by tracing back the race warnings to the application level and presenting a summarized result to the developer. The application-level race warnings can be seen as misuse warnings on shared, thread-unsafe library objects.

1.3. Filtering

To improve performance, many library classes employ advanced synchronization techniques (e.g., memory fences, spin-locks, compare-and-swap, immutability, complex locking protocols). These classes pose challenges for any static race detection and their analysis is mostly limited to model checking and verification approaches. As our analysis is aimed at application code, not library classes, we assume that libraries are correctly implemented. Thus, we use a lightweight model of their documented behavior to determine correctness. In addition, following Michael Hind's advice on the importance of client-specific pointer analysis [Hind 2001], we use this model to specialize the context sensitivity to increase precision and lower runtime.

Contributions

This paper extends the work presented at ISSTA in 2013 [Radoi and Dig 2013] (i) by providing a more formal, precise, and in-depth description of the techniques, (ii) by relaxing the IFDS lockset algorithm to allow filtering of more races, and (iii) by an algorithmic improvement that resulted in significantly better precision. In our case studies, the improvements reduce the number of race warnings by 27% on average. In one case, the number of race warnings shrank from 1735 to 2.

This paper presents the following contributions:

— **Race detection approach.** We propose three techniques aimed at making static race detection for loop-parallel code practical. Our approach (i) *specializes* in lambda-style parallel loops [lam 2015], (ii) traces, summarizes, and reports the race warnings in application code, and (iii) is aware of and uses known thread-safety properties of library classes.
— **Tool.** We implemented these techniques in a tool, ITERACE, that analyzes Java programs. We released it as open-source: http://github.com/cos/IteRace.
— **Evaluation.** We evaluated our approach by using ITERACE to analyze seven open-source projects. For context, we also analyzed the same projects with a state-of-the-art, but general, static race detection tool, JChord [Naik et al. 2006]. The results show that our specialized approach is sufficiently fast and precise to be practical. It runs in at most a few minutes and reports very few warnings for many of the case studies. In five of the seven case studies ITERACE reports only true races, i.e., it does not report any false race warning.

    We reported four of the bugs found by ITERACE to the projects' developers. One had already been known and fixed. The other three were new, and they were confirmed and fixed by the developers.

    Finally, we designed and carried out a set of experiments to measure the effect of each specialization technique, both alone and in combination with other techniques.

2. MOTIVATING EXAMPLE

To illustrate our analysis, we use a simple N-body simulation implementation, shown partially in Fig. 2; for now, only consider the code, not the extra graphical aid. An N-body simulation computes how a system of particles evolves when subjected to gravitational forces. The parallel implementation uses the loop parallelism library enhancements introduced in Java 8 [JDK 2015]. In Java 8, clients can call the parallel() method on any Collection
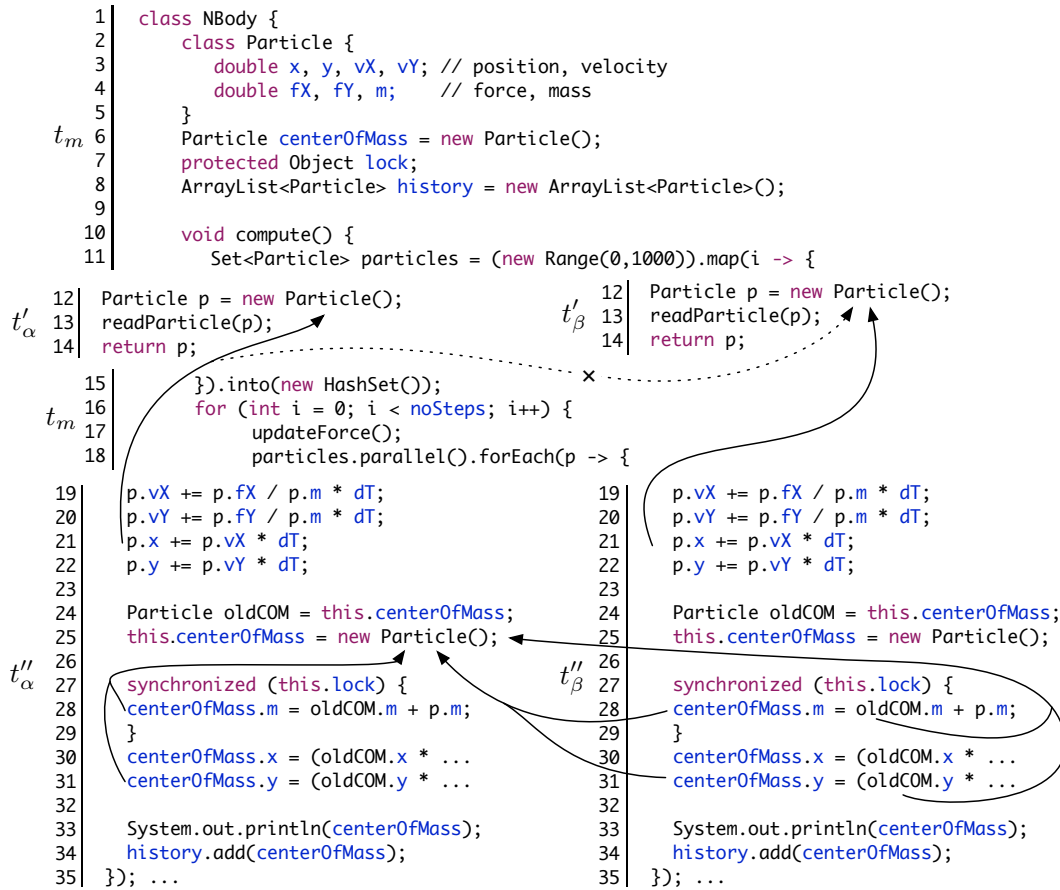
```
        1   class NBody {
        2       class Particle {
        3           double x, y, vX, vY; // position, velocity
        4           double fX, fY, m;    // force, mass
        5       }
   tm   6       Particle centerOfMass = new Particle();
        7       protected Object lock;
        8       ArrayList<Particle> history = new ArrayList<Particle>();
        9
       10       void compute() {
       11           Set<Particle> particles = (new Range(0,1000)).map(i -> {
```

```
       12   Particle p = new Particle();              12   Particle p = new Particle();
 t'α   13   readParticle(p);                    t'β   13   readParticle(p);
       14   return p;                                 14   return p;
```

```
       15        }).into(new HashSet());            ✗
  tm   16        for (int i = 0; i < noSteps; i++) {
       17            updateForce();
       18            particles.parallel().forEach(p -> {
```

```
       19   p.vX += p.fX / p.m * dT;                  19   p.vX += p.fX / p.m * dT;
       20   p.vY += p.fY / p.m * dT;                  20   p.vY += p.fY / p.m * dT;
       21   p.x += p.vX * dT;                         21   p.x += p.vX * dT;
       22   p.y += p.vY * dT;                         22   p.y += p.vY * dT;
       23                                             23
       24   Particle oldCOM = this.centerOfMass;      24   Particle oldCOM = this.centerOfMass;
       25   this.centerOfMass = new Particle();       25   this.centerOfMass = new Particle();
       26                                             26
 t''α  27   synchronized (this.lock) {          t''β  27   synchronized (this.lock) {
       28     centerOfMass.m = oldCOM.m + p.m;         28     centerOfMass.m = oldCOM.m + p.m;
       29   }                                         29   }
       30   centerOfMass.x = (oldCOM.x * ...          30   centerOfMass.x = (oldCOM.x * ...
       31   centerOfMass.y = (oldCOM.y * ...          31   centerOfMass.y = (oldCOM.y * ...
       32                                             32
       33   System.out.println(centerOfMass);        33   System.out.println(centerOfMass);
       34   history.add(centerOfMass);               34   history.add(centerOfMass);
       35   }); ...                                   35   }); ...
```

Fig. 2: **Visual representation of how our tool, ITERACE, analyses a simple N-body simulation implementation.** Each block of code is labeled with the abstract thread that executes it, e.g., t'_α. The arrows show points-to relations from variables to allocation sites, e.g., variable p at line 21 in thread t''_α may point to the abstract object instantiated on line 12 in thread t'_α. Only relevant points-to relations are shown. The dashed crossed arrow represents an abstract points-to relation that would not appear in any real execution, so it is correctly discarded in our model.

to get a "parallel view" of it. They can then execute loop-parallel operations (e.g. parallel map) by passing lambda expressions to this view.

In this example, a HashSet of particles is created by the lambda expression at lines 11-15. Then, the simulation proceeds iteratively in time steps (line 16), at each step the particles being moved according to their mass and current positions and velocities. An N-body simulation step is typically comprised of two stages. The first stage updates the forces according to the mass and current position of all particles. This stage is computed by the method updateForce, which we choose not to detail here as it is verbose and does not add value to the presentation. In the second stage, the parallel operator defined at lines 19-33 updates each particle's velocity (lines 19-20) and position (lines 21-22).

Loop: simulation.NBody.compute(NBody.java:18)

java.util.ArrayList: simulation.NBody.<init>(NBody.java:8)
 application level
    (a) simulation.NBody$2.op(NBody.java:34)
    (b) simulation.NBody$2.op(NBody.java:34)
simulation.Particle: simulation.NBody$2.op(NBody.java:25)
 .x
    (a) simulation.NBody$2.op(NBody.java:30)
    (b) simulation.NBody$2.op(NBody.java:30) [2x]
 .y
    (a) simulation.NBody$2.op(NBody.java:31)
    (b) simulation.NBody$2.op(NBody.java:31) [2x]
simulation.NBody: simulation.NBody.main(NBody.java:40)
 .centerOfMass
    (a) simulation.NBody$2.op(NBody.java:25)
    (b) simulation.NBody$2.op(NBody.java:24)
        simulation.NBody$2.op(NBody.java:25)
        simulation.NBody$2.op(NBody.java:28)
        simulation.NBody$2.op(NBody.java:30)
        simulation.NBody$2.op(NBody.java:31)
        simulation.NBody$2.op(NBody.java:33)
        simulation.NBody$2.op(NBody.java:34)

Fig. 3: IteRace's report for the example in Fig. 2

For the purpose of showing how different races are handled by our analysis, we have also included a computation of the centerOfMass of all particles (lines 24-31). Also, lines 33-34 print and then log the movement of the center of mass in the ArrayList history.

The center of mass is stored in an instance field of NBody (line 6). The computation proceeds as follows. Line 24 stores the current value of the centerOfMass field in a local variable oldCOM. Then, the centerOfMass field is updated to a new Particle object (line 25) which is populated with values based on the oldCOM and the current particle, p (lines 27-31). As this computation is part of the parallel operator, there are multiple threads executing this code concurrently. The NBody object is shared between these threads, so there are multiple races that can occur on the centerOfMass field and Particle object referred by it. The centerOfMass field write on line 25 can race with another thread executing the instruction on line 25 or any of the read field instructions at lines 24, 28, 30, or 31. Also, lines 28, 30 and 31 write and read fields of the Particle referenced by centerOfMass. This is the object initialized at line 6 but it is not thread-local, so multiple threads could access the same Particle. The accesses to fields x and y (lines 30 and 31) are not synchronized so they are racing. The accesses at line 28 are protected by a unique lock shared between all threads, so they are safe.

Next, line 33 prints the current centerOfMass. Although this action accesses shared resources, i.e., the standard output stream, it is safe due to synchronization within the PrintStream class.

Finally, line 34 logs the current center of mass into an ArrayList pointed to by the history field of the NBody object. As the history collection is shared and the ArrayList class is not thread-safe, there will be races on the inner state of ArrayList.

Fig. 3 shows IteRace's output for our example program. The first line is the parallel loop which contains the races. Our tool then groups the reports by the allocation site (i.e., object instantiation site) of the object involved in the race. We have three groups of races: on the history ArrayList object instantiated at line 8, on the centerOfMass Particle object

instantiated at line 25, and on NBody object itself instantiated in the main method of the program (not shown). Each group of races is further grouped by the affected field, and all application-level races are bundled together.

We first consider the group of races on the history ArrayList, which describes the "application level" race at line 34. IteRace reports this race as the concurrent execution of the instruction identified by the next line, labeled "(a)", and the instruction identified by the following line, labeled "(b)".

The next group of races, on the centerOfMass object, is further divided into two subgroups based on the affected field. The "[2x]" label says that there are two accesses on that line that are involved in the race. For the race on the x field at line 30, we have a write access to the field, labeled "(a)", and two read accesses under the "b" label. Thus, the entire group has four races, two on field x and two on field y.

The last group of races, on the centerOfMass *field* of the NBody object, describes the possible concurrent execution of the write access at line 25, labeled "(a)", and several read/write accesses executed by a different thread, listed under the "(b)" label.

The next section explains how IteRace correctly identifies all of the 12 races described above, and how it filters out false positives. The *Filtering* phase eliminates the races on the standard output while the *Bubble-up* transforms the race warnings in the ArrayList to a single warning on line 34. Finally, *Synchronized* determines that a race cannot occur at line 28 because the accesses are protected by the shared lock. Furthermore, the accesses on fields vX, vY, x, and y at lines 19-22 are not races and IteRace does not report them as such. In this case, an analysis lacking *2-Threads* and relying on escape analysis would report false warnings.

3. RACE DETECTION

We now explain how IteRace represents programs, how it detects races, and how it avoids false warnings.

Figure 4 presents a high level overview of IteRace. WALA [wal 2015] provides the underlying Andersen-style static pointer analysis. The call graph is computed on-the-fly along with the heap model, based on a specialized context sensitivity. Each of our techniques specializes the context sensitivity, as detailed in sections 3.1, 3.2, and 3.3. The analysis is flow-insensitive, with the exception of the limited amount of flow sensitivity provided by static single assignment. Objects are abstracted by allocation sites and fields are distinguished. Method calls have a bounded context sensitivity that is specialized by each technique. On completion, the pointer analysis produces a static call graph representing the execution, a control-flow graph for each method, and a heap graph.

Next, IteRace computes the set of potential races (pairs of accesses that would race if not synchronized) by traversing the program representation and matching instructions using alias information from the heap graph (Sec. 3.1).

Also, for each statement in the program, IteRace computes the lock set that protects it. This is achieved by an IFDS analysis [Reps et al. 1995].

Then, the *Filtering* phase (Sec. 3.2) eliminates races based on *a priori* thread-safety information for classes.

Accesses protected by the same lock are race-free. The *Deep-Synchronized* phase (Sec. 3.5) filters out the potential races on such accesses, yielding the set of actual races.

Then, IteRace "bubbles up" the races that occur in library code and reports them in application code, on the library-method calls that led to them (Sec. 3.3).

Finally, *Synchronized*, a stage similar to *Deep-Synchronized*, further prunes the bubbled-up race warnings.
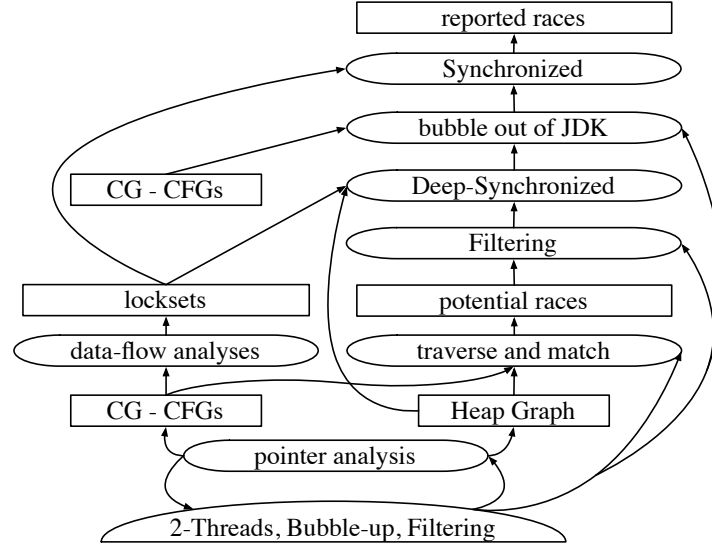
Fig. 4: **Analysis overview.** Ovals represent different sub-analyses. Rectangles represent intermediate and final data structures. The bottom half-oval represents the specialized context sensitivity mechanism.

<div align="center">Domains:</div>

(method, instruction)	$m \in \mathbb{M},\ i \in \mathbb{I}$
(object)	$o \in \mathbb{O} = \mathbb{N} \times \mathbb{I}$
(class object)	$o_S \in \mathbb{O}_S \subset \mathbb{O}$
(parallel collection)	$o_c \in \mathbb{O}_c \subset \mathbb{O}$
(loop)	$l \in \mathbb{L} \subset \mathbb{O}_c \times \mathbb{N} \times \mathbb{I}$
(thread)	$t \in \mathbb{T} = (\mathbb{L} \times \{\alpha, \beta\}) \cup \{t_m\}$
(the main thread)	$t_m \in \mathbb{T}$
(an α thread; similar for β)	$t_\alpha \in \mathbb{T}_\alpha = \mathbb{L} \times \{\alpha\} \subset \mathbb{T}$
(call graph node)	$n \in \mathbb{N} = \mathbb{X} \times \mathbb{M}$
(boolean flag)	$b \in \mathbb{B}$
(CG node context)	$x \in \mathbb{X} = (\{t_m\} \times 2^{\mathbb{I}}) \cup ((\mathbb{T}_\alpha \cup \mathbb{T}_\beta) \times \mathbb{B} \times [\mathbb{B}] \times \mathbb{O})$

<div align="center">Notation:
If $\gamma = \langle \gamma_1, \ldots \gamma_k, \ldots \rangle \in \Gamma_1 \times \cdots \times \Gamma_k \times \ldots$
then $\pi_{\Gamma_k}(\gamma) = \gamma_k$ is the projection of γ on the Γ_k domain</div>

<div align="center">Fig. 5: **Program model**</div>

$$\text{context} : \mathbb{N} \times \mathbb{M} \times \mathbb{I} \times [\mathbb{O}] \to \mathbb{X}$$

$$\text{context}(\langle x_c, m_c \rangle, m, i, O_p) =$$

$$= \begin{cases} \langle t_m, (O_p \cap O_c) \cup \pi_2(x_c) \rangle & \text{if } \pi_{\mathbb{T}}(x_c) = t_m \text{ and outside loop} \\ \langle t, \text{false}, [], \epsilon \rangle & \text{if } \pi_{\mathbb{T}}(x_c) = t_m \text{ and entering } t \in \mathbb{L} \times \{\alpha, \beta\} \\ \langle t, x_{tts} \vee x'_{tts}, x'_s, o'_r \rangle & \text{if } x_c = \langle t, x_{tts}, x_s, o_r \rangle \end{cases}$$

$$\text{where}$$

$x'_{tts} = \text{true if } \textit{transitively-thread-safe}(m)$

$x'_s = [t_o = t_m \vee \pi_{\{\alpha, \beta\}}(t_o) \neq t \,|\, o_k \in O_p \text{ and } t_o = \pi_{\mathbb{T}}(\pi_{\mathbb{X}}(\pi_{\mathbb{N}}(o_k)))]$

$o'_r = O_p(1)$, if the receiver object $O_p(1)$ is a container, e.g., a collection

$$\begin{aligned} \langle x_c, m_c \rangle &\quad - \text{ caller CG node} \\ m &\quad - \text{ callee method} \\ i &\quad - \text{ call site instruction} \\ O_p &\quad - \text{ set of actual parameters, including the receiver object} \\ O_c &\quad - \text{ set of parallel collections in the program} \end{aligned}$$

Fig. 6: **Context sensitivity**

3.1. 2-Threads program model

Figure 5 presents the way ITERACE models the program. $\mathbb{M}, \mathbb{I}, \mathbb{O}$ are domains that represent methods, instructions, and objects. Objects are modeled as allocation sites in call graph nodes. For uniformity, classes are represented as abstract objects, $\mathbb{O}_S$, and static fields as fields of these objects.

The main thread of the program is modeled by an abstract thread $t_m \in \mathbb{T}$ (lines 1-8 and 16-17 in our example). As outlined in Fig. 1, the concrete threads executing each loop $l \in \mathbb{L}$ are modeled by two abstract threads, t_α and t_β. In our example (Fig. 2), $\langle t'_\alpha, t'_\beta \rangle$ and $\langle t''_\alpha, t''_\beta \rangle$ model the threads executing the parallel loops at line 11 and 18, respectively. We will further use the notation $t : x$ to refer the instructions at line number x as executed in the context of abstract thread t; e.g., $t'_\alpha : 12$ refers the instruction at line number 12 executed by t'_α.

Each call graph node $n \in \mathbb{N}$ is named by the called method $m \in \mathbb{M}$ and a context $x \in \mathbb{X}$. The context is determined using the context function detailed in Fig. 6. The abstract thread $t \in \mathbb{T}$ executing the method is part of the context, along with other information explained in the next subsections.

Figure 6 shows how ITERACE determines the call-context sensitivity. Let $\langle x_c, m_c \rangle$ (context and method) be a call graph node. It contains an instruction i which calls method m with the list of arguments O_p. The context for the call graph node created by this call is given by $\text{context}(\langle x_c, m_c \rangle, m, i, O_p)$. ITERACE computes the context in three different ways depending on the executing thread of the caller and callee. In all cases, the context contains the executing thread.

In the first case the main thread executes $\langle x_c, m_c \rangle$, and m is not a loop operation (i.e., the computation is not entering a loop). In this case, ITERACE refines the context with the set of all arguments on the call graph which are parallel collections (O_c). The set is the union of all current arguments which are parallel collections ($O_p \cap O_c$) with the set of such arguments from the calling call graph node x_c (x_c is in this case a pair, with its second element being the parallel collection set, so we do a set union with its projection on the second element). The context sensitivity on the parallel collection abstract objects avoids the conflation of

$$
\begin{array}{ll}
\text{c.forEach(op)} & \left|\begin{array}{ll} \mathrm{op}(e_\alpha) & [t_\alpha] \\ \mathrm{op}(e_\beta) & [t_\beta] \end{array}\right. \\[2em]
\text{c.map(op)} & \left|\begin{array}{ll} e_\alpha = \mathrm{op}(e_\alpha) & [t_\alpha] \\ e_\beta = \mathrm{op}(e_\beta) & [t_\beta] \\ return\ c & [t_m] \end{array}\right. \\[3em]
\text{c.reduce(base, op)} & \left|\begin{array}{ll} x_1 = \mathrm{op}(e_\alpha, \mathrm{base}) & [t_\alpha] \\ x_2 = \mathrm{op}(x_1, e_\beta) & [t_\beta] \\ return\ x_2 & [t_m] \end{array}\right.
\end{array}
$$

Fig. 7: **Model of collection operations.** The abstract thread executing each operation is bracketed to its right.

abstract parallel loops iterating over distinct parallel collections. The conflation led to an inflation in the number of false warnings in one of our case studies.

In the second case the computation is just entering a loop operation executed by thread t, which, in our model, is either the α or the β thread. This is the point where the context structure is transformed from the simple thread-and-collection pair used outside loop to the quad used inside the loop. The quad is initialized with the current thread and empty values $(\langle t, \mathrm{false}, [], \epsilon \rangle)$.

The third case represents a call between two methods executed within the loop operation. The context is a quad containing:

— the executing thread (t_α or t_β).
— a boolean flag, x_{tts}, marking whether the current node is *transitively-thread-safe* (explained in Section 3.2).
— x_s, a list of boolean flags corresponding to the sharing nature of the arguments (explained at the end of Section 3.2).
— o_r, a reference to the receiving object if it is a container (e.g., a collection). This adds a level of container context sensitivity (see [wal 2015]), which helps with not conflating abstract objects which pass through collections.

The analysis matches loops operating on the same collection ($\langle t'_\alpha, t'_\beta \rangle$ and $\langle t''_\alpha, t''_\beta \rangle$ in our example) using the abstract object characterizing both loops (see loop in Fig.5). When the abstract object represents multiple concrete objects that are not all processed by the loop, the analysis might introduce spurious warnings, although it would still be safe. The context sensitivity on the parallel collection described for the first case above helps alleviate this effect by precisely tracking the collections of interest through the program.

The analysis maintains a special model for each collection of interest. The elements of a collection are modeled by two abstract fields, e_α and e_β. Fig. 7 shows how each of the abstract threads, t_α and t_β, processes one of the abstract fields e_α and e_β, respectively. This model allows our technique to distinguish between elements processed by different threads. For example, in the case of the forEach operation, different elements of the collection, e_α and e_β, are processed by different threads, t_α and t_β, respectively. The model is capable of distinguishing that processing e_α only updates e_α, not both e_α and e_β, and vice-versa. While our implementation does not cover all the new Java 8 collection operations [lam 2015], it can be easily adapted to do so once the specification stabilizes.

The above modeling is used for both the parallel and the sequential loop operations over the collection of interest. This allows ITERACE to understand the relationships between the elements of the collection as it is processed by different loops. In Figure 2, both the

collection initialization at lines 11-15 and the processing at lines 18-35 are modeled. Thus, ITERACE sees that the element p in t''_α is the same as p in t'_α but different from p from t'_β.

Definition 3.1. A *potential race* is a pair of accesses $(\langle n_\alpha, i_\alpha \rangle, \langle n_\beta, i_\beta \rangle)$ to the same field of the same object, such that one is a write access executed by a t_α (i.e., $\pi_\mathbb{T}(\pi_\mathbb{X}(n_\alpha)) = t_\alpha$) and the other is either a read or a write executed by t_β (i.e., $\pi_\mathbb{T}(\pi_\mathbb{X}(n_\beta)) = t_\beta$), with and t_α and t_β modeling the same loop (i.e., $\pi_\mathbb{L}(t_\alpha) = \pi_\mathbb{L}(t_\beta)$).

In our example, there are several potential races on the centerOfMass field of the NBody object. Instruction $t''_\alpha : 25$ writes the field centerOfMass while instructions $t''_\beta : 24$ and $t''_\beta : 25$ read and respectively write the same field of the same object. Therefore, according to the definition above, the pairs of accesses $\langle t''_\alpha : 25, t''_\beta : 24 \rangle$ and $\langle t''_\alpha : 25, t''_\beta : 25 \rangle$, on the centerOfMass field of the NBody object are potentially racing. Accesses at lines 28, 30, and 31 in thread t''_β are also racing with instruction $t''_\alpha : 25$ because they read centerOfMass.

The more interesting cases are the potential races on fields of the Particle references by centerOfMass. We will look at the write access at $t''_\alpha : 31$ and the read/write accesses at $t''_\beta : 31$. centerOfMass at $t''_\alpha : 31$ may point to the objects instantiated at either of $t_m : 6$ (the pointer analysis is flow-insensitive), $t''_\beta : 25$ or $t''_\alpha : 25$. centerOfMass and oldCOM at $t''_\beta : 31$ may point to the same three objects. For the latter of the objects, i.e., the one instantiated at $t''_\alpha : 25$, there are two potential races on its y field, one for the write-write accesses (both writes on centerOfMass), and one for the write-read accesses (write on centerOfMass, read on oldCOM). Similarly, there are two potential races for each of the objects instantiated at $t_m : 6$ and $t''_\beta : 25$. It is not possible for a race to occur on the object instantiated at $t_m : 6$ but ITERACE is flow insensitive so it does not take into consideration that the field update at line 25 happens before the potential race on line 31. Still, the resulting false warnings are not particularly distracting to the programmer as they are usually accompanied by warnings of real races on the same variable, as in our example. Also, section 6 shows how the way we report races makes such cases less of a nuisance.

We now look at accesses that are not potential races because of our particular representation of collection operations, i.e., two abstract threads for each operation with an underlying modeling of the collection elements. Let us consider the pair of non-racing write accesses to p.x $\langle t''_\alpha : 21, t''_\beta : 21 \rangle$. They are not racing as each refers to a different unique element of the collection.

In order to determine if they are racing, an analysis needs to determine whether the p variables from each of the threads may alias. If the parallel loop iteration would be modeled by only one abstract thread, there would be only one abstract representation for the p variable so it would obviously may-alias. Then, thread escape analysis could be employed to cut down the number of accesses that can be involved in a race. In this case, escape analysis would not solve the problem as the object referenced by the variable is escaping through particles. Then, other more expensive analyses could be further employed to refine the results, for example [Naik and Aiken 2007].

In contrast, our approach is simpler yet very effective, making thread-escape analysis unnecessary. As ITERACE models each parallel loop by two threads, it does not need to consider races that might occur between instructions of the same abstract thread. Also, as ITERACE models the collection to distinguish between the elements processed by each of the two abstract threads, it achieves collection-element sensitivity. For example, the object initialized at instruction $t'_\alpha : 12$ is identified as the same as the object accessed at $t''_\alpha : 21$, but different from the object initialized at $t'_\beta : 12$ (crossed arrow). Similarly, the object initialized at instruction $t'_\beta : 12$ is the same as the object accessed at $t''_\beta : 21$ and different from the one at $t'_\alpha : 12$. Hence, p at $t''_\alpha : 21$ and p at $t''_\beta : 21$ may not alias, therefore $\langle t''_\alpha : 21, t''_\beta : 21 \rangle$ cannot race.

Additionally, all objects are labeled with their instantiation thread. ITERACE uses this information to alleviate the effect of the pointer analysis not being meet-over-all-*valid*-paths [Sharir and Pnueli 1981]. The code listing below shows a very simple example of how a shared object can "piggyback" on a non-shared object's abstract path through the program and then introduce a false race. Without any extra context sensitivity, both calls to returnMyself are represented by the same call graph node. Thus, particle points to both the objects referenced by sharedParticle and the new, locally initialized Particle. As the pointer analysis does not filter invalid paths, p will also point to both the new object, as it should, and the shared object. Now, any write access, like the one to the x field below, will introduce false warnings.

```
public void returnMyself(Particle particle) {
    return particle;
} ...
returnMyself(sharedParticle);
Particle p = returnMyself(new Particle());
p.x = 7;
```

To alleviate this effect, our tool makes calls within parallel iterations context sensitive on the sharing nature of their arguments. Each call graph node executed by a loop operator has in its context (as part of $x \in \mathbb{X}$ – see Fig. 5) a list of boolean flags X_s with $X_s(k) = $ true meaning that the k argument has not been instantiated in the current iteration (see x'_s in Fig. 6). For the above example, $X_s(1)$ is true for the call on sharedParticle but false for the call on the new Particle. Thus, two distinct call graph nodes are created for returnMyself. In effect, p only points to the new object, and no false races are introduced.

3.2. Filtering using a thread-safety model

ITERACE uses a simple *a priori* (mostly provided by ITERACE and completed by the user when needed – see Section 4.3) thread-safety model of the classes to drastically reduce the number of warnings introduced by the intricate thread-safety mechanisms in libraries. For this purpose, it adjusts the context sensitivity and adds one warning-filtering phase.

Filtering uses the following *a priori* information about methods. A method of a class:

— is ***thread-safe*** if the instructions of this method cannot race in any concrete invocation.
— is ***transitively-thread-safe*** if it is *thread-safe* and any other invocation reachable from its invocation cannot be involved in races. This category of methods includes, but is not limited to, methods of immutable classes. All *transitively-thread-safe* methods are *thread-safe* (by definition). The converse is not true, as explained at the end of this subsection.
— ***instantiates-only-safe-objects*** if any object instantiated inside the method, but not necessarily in other methods called by it, is thread-safe. This property is mostly useful for anonymous classes as they cannot be modeled with *thread-safe* because there is no class name to hook to.
— ***circulates-unsafe-objects*** if the method may either return or receive a possibly non-thread-safe object as a parameter.

Using this information, the context of a callee is generated from the context of the caller by adding a *transitively-thread-safe* sticky flag when the callee is *transitively-thread-safe*, as shown in Fig.6. The flag is sticky in the sense that they will be propagated downstream unless explicitly removed.

The *Filtering* stage uses the above model and the generated flags to filter out accesses that cannot be involved in races. An access in the abstract invocation n_a of method m_a on object o instantiated in a method m_o, cannot be involved in a race if any of the following conditions is met:

— *thread-safe*(m_a)
— *instantiates-only-safe-objects*(m_o)
— *transitively-thread-safe*(n_a)

It is possible to have methods that are *thread-safe* but not *transitively-thread-safe*. Let us go back to the example in Fig.2. Line 34 contains a call to PrintStream on the method println(Object) listed below:

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

This method is *thread-safe* as a race cannot occur within it but it is not considered *transitively-thread-safe* because of the call to String.valueOf. This method verifies whether the passed object is a String and calls toString on it otherwise. The problem is that we know nothing about the thread-safety of toString on arbitrary objects. Even protecting String.valueOf(x) within a synchronized section would not help, since it could still race with another access holding a different or no lock. The method also calls print(String) and newLine(). These methods are *transitively-thread-safe* as they are also synchronized internally and do not operate on any object supplied from outside.

3.3. Bubble-up to application level

Next, ITERACE bubbles up the races that occurred in libraries to application level. The intuition is that the application programmer does not care which library inner object the accesses occurred on. She only cares which accesses to said application-level object generate races. For line 34 in our example (Fig. 2), the programmer doesn't care that the races occurred on fields elementData and size inside the ArrayList object. She only cares about the pair of accesses on history. The programmer can tell ITERACE which classes to consider as library classes, yielding reports at various depth levels.

Reporting a race means reporting a racing pair of accesses. ITERACE reports each of the accesses occurring in library code as a set of method invocations in application code that lead to the in-library access.

$\langle n^a, i^a \rangle$ is an *application-level access leading to* $\langle n, i \rangle$ iff the method associated with call graph node n^a is not *is-library*, and there exists a path from n^a to n for which all intermediate nodes except n^a are through library code. The set of application-level accesses leading to a racing access is computed by traversing the call graph backwards, from the race to the first call graph node outside of library code.

For each race $\langle \langle n_\alpha, i_\alpha \rangle, \langle n_\beta, i_\beta \rangle \rangle$ in library code, ITERACE creates the set of application-level races by matching the application-level accesses leading to $\langle n_\alpha, i_\alpha \rangle$ with the application-level accesses leading to $\langle n_\beta, i_\beta \rangle$. The accesses are matched based on their application-level receiver objects.

ITERACE also adds a layer of object sensitivity [Milanova et al. 2005] on calls to methods of objects known to be containers of other objects (e.g., JDK collections). This improves the precision of the *Bubble-up* technique as it induces a better separation between application and libraries.

Interactive iterative refinement. In addition to not reporting race warnings in library code, the *Bubble-up* technique is also used for narrowing down the cause of a race. The programmer first aggressively marks application classes as library code in order to make the analysis report warnings much closer to the loop body. This drastically reduces the

number of warnings but also hides the reason the analysis considers some pairs of accesses as leading to races. Then, the programmer gradually removes the *is-library* markings until the source for the race reveals itself. After each analysis run, for each race or group of related races, the programmer asks herself the question: "Why are these accesses racing?". If she cannot find an answer by inspecting the code, she removes the library marking for the offending accesses, and reruns the analysis. She repeats this process until all the reported race warnings are fully understood.

3.4. Abstract locksets

In order to determine if a program is correctly synchronized, one needs to determine which locks protect each instruction that may run in parallel with other instructions. In the case of a static analysis such as ours, a conservative set of locks needs to be determined.

In Java, a lock can be either an instance of a class or the class itself. We represent instance locks by the SSA variable that was dereferenced when acquiring the lock in a particular call graph node (i.e., by a pair $\langle n_k, v_k \rangle \in \mathbb{N} \times \mathbb{V}$), and class locks by the corresponding abstract class object (an $o_S \in O_S$).

Let k be an abstract lock. k *protects* the statement $\langle n, i \rangle$ if, in all possible executions, lock k is held by thread $\pi_{\mathbb{T}}(n)$ at $\langle n, i \rangle$. We compute the *protects* property for a particular thread by encoding it as an interprocedural, finite, distributive, subset (IFDS) problem [Reps et al. 1995].

For thread t (t_α or t_β in our case), $\text{protects}_t(\langle n, i \rangle)$ gives the set of locks protecting statement $\langle n, i \rangle$. It is computed as the set difference $\text{allLocks}(t) \setminus \text{notProtects}(\langle n, i \rangle)$, where $\text{allLocks}(t)$ is the set of all abstract locks acquired by thread t and $\text{notProtects}(\langle n, i \rangle)$ is the solution at node $\langle n, i \rangle$ for the IFDS problem $IP_t = (G^*, L, F, M, \cup)$.

G^* is the supergraph of the program, i.e., the graph obtained by replacing each node n of the call graph with its corresponding control-flow graph (i.e. the CFG of $\pi_{\mathbb{M}}(n)$), each CFG node i in turn being replaced by the pair $\langle n, i \rangle$. There are interprocedural arcs between n's predecessors and the entry node of n's new CFG, and between this CFG's exit nodes and n's successors.

L is the domain of the dataflow problem. In our case, it is the set of locks that could be held by thread t. As our problem models the notProtects property, node $\langle n, i \rangle$ having the value L, i.e. the entire domain, means it is not protected by any lock.

Our set of distributive dataflow functions is:

$$F = \{\text{acquireLock}_k, \text{releaseLock}_k \mid k \in L\} \subset 2^D \to 2^D$$
$$\text{acquireLock}_k(l) = l \setminus \{k\}$$
$$\text{releaseLock}_k(l) = l \cup \{k\}$$

$M : (\mathbb{N} \times \mathbb{I}) \times (\mathbb{N} \times \mathbb{I}) \to F$ is the mapping between G^*'s edges and the dataflow functions:

$$M(\langle n, i \rangle, \langle n_s, i_s \rangle) = \begin{cases} \text{acquireLock}_{\langle n, v \rangle} & \text{if } n = n_s \wedge i \text{ acquires a lock on } v \\ \text{releaseLock}_{\langle n, v \rangle} & \text{if } n = n_s \wedge i \text{ releases the lock from } v \\ \text{acquireLock}_{\langle n_s, v_{this} \rangle} & \text{if } n \neq n_s \wedge i_s \text{ is a CFG entry node } \wedge \\ & \quad \wedge \pi_{\mathbb{M}}(n_s) \text{ is a \textbf{synchronized} method} \\ \text{releaseLock}_{\langle n, v_{this} \rangle} & \text{if } n \neq n_s \wedge i \text{ is a CFG exit node } \wedge \\ & \quad \wedge \pi_{\mathbb{M}}(n) \text{ is a \textbf{synchronized} method} \\ \text{acquireLock}_{\langle n_s, o_C \rangle} & \text{if } n \neq n_s \wedge i_s \text{ is a CFG entry node } \wedge \\ & \quad \wedge \pi_{\mathbb{M}}(n_s) \text{ is a \textbf{synchronized} method} \\ & \quad \text{of class object } o_C \text{ (i.e., it is \textbf{static})} \\ \text{releaseLock}_{\langle n, o_C \rangle} & \text{if } n \neq n_s \wedge i \text{ is a CFG exit node } \wedge \\ & \quad \wedge \pi_{\mathbb{M}}(n) \text{ is a \textbf{synchronized} method} \\ & \quad \text{of class object } o_C \text{ (i.e., it is \textbf{static})} \end{cases}$$

Our approach is similar to [Naik et al. 2006] but we choose to represent locks as variables in call graph nodes, not as a subset of the abstract objects from the abstract heap graph. Another differentiating aspect is that we express the lockset problem as an IFDS problem instead of a graph traversal. Graph traversal is a *meet-over-all-paths* solution, while the IFDS framework gives a more precise, *meet-over-all-valid-paths*, solution [Reps et al. 1995]. The precision advantage is obtained without enumerating all paths so the analysis speed penalty is generally low. Furthermore, the IFDS-based approach allows ITERACE to analyze not only synchronization which uses **synchronized** blocks and methods, but also synchronization which uses Java classes implementing the Lock interface. Using IFDS for computing locksets is mentioned by [Qi et al. 2009] but they do not explain their implementation or encoding.

3.5. Synchronized accesses

ITERACE can now filter the races based on the abstract lockset information. A race $\langle\langle n_\alpha, i_\alpha\rangle, \langle n_\beta, i_\beta\rangle\rangle$ is filtered out if the intersection of the sets of abstract objects pointed to by their abstract locksets is not empty:

$$\{o\,|\,\exists\,k \in \text{protects}(n_\alpha, i_\alpha)\,.\,o \in \text{points-to}(k)\} \cap \{o\,|\,\exists\,k \in \text{protects}(n_\beta, i_\beta)\,.\,o \in \text{points-to}(k)\} \neq \emptyset$$

We filter safe accesses at two levels: once on an initial set of races, as in previous work [Naik et al. 2010], and once after the *Bubble-up*. Our evaluation revealed that applying the algorithm after *Bubble-up* is slightly faster and more effective. The reason lies in the library objects' abstraction imprecision. A single call graph node of a library method abstracts multiple runtime invocations. When invocations that are protected by application-level synchronization are conflated with unprotected invocations, and locksets are checked at library level, all accesses are considered unsafe. If the accesses are checked at application-level, the tool has better chances of distinguishing safe accesses.

4. DISCUSSION

In this section we discuss ITERACE's safety, speed, usability, and reusability.

4.1. Soundness

ITERACE is subject to the typical sources of unsoundness for static analysis, i.e., it has only limited handling of reflection and native method calls, to the extent provided by WALA.

The *Synchronized* phase unsafely uses may-alias information to approximate must-alias lock relations. The analysis can easily be adapted to use a must-alias analysis once a scalable must-alias analysis is available. Also, our evaluation shows that the *Deep-Synchronized* and *Synchronized* phases have much less warning-reduction effect than the others. The programmer can choose to deactivate these phases to get safer results.

The *Filtering* technique relies on the programmer specifying which methods and classes are *thread-safe*, *transitively-thread-safe*, *instantiates-only-safe-objects*, or *circulates-unsafe-objects* (Section 3.2). An incorrect specification may lead the analysis to miss true races. We have already specified the thread-safety characteristics of a large number of JDK classes and methods by using the javadocs as a guide. A programmer using ITERACE may need to extend this if she uses other libraries containing thread-safe classes.

4.2. Monitor structure

ITERACE is designed to analyze the lambda-style loop-parallel parts of the program and cannot reason about concurrency that appears by spawning other threads besides the ones used by the parallel loops. Nested parallel loops are also not supported. In such cases, ITERACE warns the programmer about the potentially unsafe thread spawn. Extending our tool to handle other concurrency constructs should be straightforward. The *Bubble-up* and *Filtering* techniques could be applied directly and would be beneficial. *2-Threads* would

not be applicable directly but its underlying idea could prove useful in designing similar techniques for other thread structures. Also, our analysis does not handle cases where the parallel collection is updated outside collection operators, e.g., using explicit loops.

4.3. Programmer effort

The *Bubble-up* and *Filtering* techniques require some input from the programmer regarding the library and thread-safety characteristics of classes and methods. The specifications are not mixed with the code but are kept in separate files as regular expressions matching class and method names. ITERACE already contains specification files for commonly used classes.

Bubble-up asks the programmer to mark the classes that should be considered as library code. These markings can also be used for understanding the fault leading to the reported races – Section 3.3 gives a description of this process and Section 6 evaluates the programmer effort involved. We do not believe the process can be significantly automated because most of the developer time is not spent interacting with the tool, but understanding the analyzed code. The gradual unfolding (marking fewer and fewer classes as library code) is a way for the developer to understand the fault underlying the race warning (or, of course, to understand that it is actually a false warning). While improvements could be made to the user experience (e.g., precomputing the results of running the tool using various library-marking depths so that the developers would not have to wait for results), and techniques such as delta debugging may offer speed-ups, we do not envision a complete automation of this process of understanding.

As mentioned in Section 4.1, *Filtering* relies on thread-safety specifications of library classes which are given by the programmer. We have already specified the characteristics of many JDK classes. Our specifications are based on the documented behavior of the classes (e.g., ConcurrentHashMap is documented as being thread-safe and we mark it as such). We assume the documented behavior is correct. The effort is minimal as the specifications are in most cases a direct translation of the documentation.

The current thread-safety specification files have, cumulatively, around 120 short regular expressions (generally, each regular expression matches one package, one class, or one method – e.g., "java.util.concurrent.ConcurrentHashMap.*" marks the entire ConcurrentHashMap class as thread-safe). Just five of the regular expressions are specific to the applications used in our evaluation; the rest match JDK code. Thus, in our experience, the programmer needs to add extra thread-safety markers in very few cases.

4.4. Optimization

An early implementation of ITERACE was severely slowed down by the data structure storing the set of races. In the early implementation, the races were simply stored as a set of pairs of accesses. As the number of potential races, i.e. before being filtered and before flattening the context sensitivity, can be very large (up to a few millions in our experiments), the set storing them consumed large amounts of memory and iterating over the set for filtering was very slow. The solution is to store the races in a specialized, hierarchical, data structure. The structure is similar to the way ITERACE reports races, explained in Section 2. The races are grouped by loop, then by object, and finally by field. The final subgroup of races is then recored not as a set of pairs of accesses but as a pair of sets of accesses, with the interpretation that any access in the first set may race with any access in the second set. This structure not only has a smaller memory footprint, but is cheaper to construct (no set membership checks) and it speeds up filtering. E.g., if we know that a particular object is thread-safe, we can simply remove all groups of races on that object without iterating over them. Furthermore, ITERACE tries to filter races as early as possible, e.g., races in thread-safe methods are simply not added to the race set, instead of being removed afterwards.

4.5. Reuse

While *2-Threads* is specific to parallel programs, *Filtering* and *Bubble-up* may be of interest to other race detector developers. *Filtering* can be easily adapted to other thread structures, and the cost of marking the thread-safety properties of library classes is not high, with most of the markings (e.g., JDK) shared among analyzed applications (see Section 4.3). The iterative approach based on *Bubble-up* can be adapted to other contexts where the programmer has to understand complex interactions revealed on deep static call graphs.

5. EVALUATION METHODOLOGY

We evaluate our tool by answering the following questions:

(1) **Is ITERACE practical?** As the main shortcoming of static race detection is the high number of warnings, we gauge practicality by the number of warnings the programmer has to inspect. Precision is also important so we also check how many of the reported warnings lead to true races. For context, we also compare our tool with a state of the art, but general, data race detection tool for Java, JCHORD[Naik et al. 2006].
(2) **What is the impact of each specialization technique?** For each specialization technique we analyze how much it reduces the number of warnings and how it affects runtime. We measure each specialization technique as applied individually and in combination with other techniques.

Table I: **Evaluation suite.** Column 2 shows in parentheses which part of the application has been parallelized. Column 3 shows the source lines of code (i.e., without comments or blank lines) for the application and its libraries. Column 4 shows the number of methods analyzed by ITERACE. The size of library code varies as some applications use extra libraries besides JDK. The number of methods reflects methods reached by the race detector.

Project	Description (parallel section)	SLOC (k) (app+lib)	# methods
MC	Monte Carlo simulation (the separate simulations) [Bull et al. 1999]	1.4 + 220	252
Em3D	3D EM wave propagation simulation (force update) [Cahoon and McKinley 2001]	0.2 + 220	80
Coref	NLP Coreference finder (processing documents) [Bengtson and Roth 2008]	41 + 225	927
WEKA	data mining software (generation of clusterers) [Hall et al. 2009]	301 + 253	1236
Lucene	Lucene search benchmark (separate searches) [Blackburn et al. 2006]	48 +220	2363
JUnit	testing framework (JUnit's own test suite)	16 + 220	508
CIlib	computational intelligence library (simulation engine)	53 + 454	1957

We evaluate our approach by using ITERACE to analyze the 7 open-source Java projects shown in Table I. Then, we use JChord to analyze the same projects under the same condi-

tions and compare the results. Finally, we measure the impact of each of our specialization techniques.

5.1. Case studies

When building the evaluation suite, we first looked for applications with parallel implementations that used loop-parallelism. Unfortunately, the lack of a proper loop parallelism library in JDK has discouraged programmers from parallelizing their programs. We have only found three applications where programmers have used a form of loop parallelism to improve the performance of their application, i.e., Lucene, JUnit, and CIlib. Thus, we looked further to applications that have sequential implementations but where the underlying algorithm is inherently parallel and included four more applications, i.e., MonteCarlo, Em3D, Coref, and WEKA.

The evaluation suite is heterogenous: it has applications from different domains (benchmarks, NLP, data mining, computational intelligence, testing) and of various sizes, from hundreds of lines of code to hundreds of thousands. Table I shows a short description of each application and indicates which part of it is parallel, the application's size in lines of code, and the number of methods analyzed by our tool.

As Java 8 has only been recently released, analysis tools, including WALA, do not yet have support for its new features, in particular for lambda expressions. In Java, anything that can be expressed through lambda expressions can also be expressed, more verbosely, using anonymous classes. For evaluation purposes, we created a collection-like class based on ParallelArray [Par 2015] that exposes part of the new collection methods introduced in Java 8, but implemented with anonymous classes (e.g., .forEach(p $->$ { ... }) becomes .forEach(**new** Procedure<Particle>() { **public void** op(Particle p) {...} })). Once WALA handles lambda expressions, adapting the implementation will be trivial.

For already-parallel applications, we manually adapted the implementation to use our collection. We changed the original implementations as little as possible, i.e., we neither performed any additional refactoring, nor fixed any races.

For the sequential applications, we parallelized each of them by performing the following steps:

(1) run the YourKit [You 2015] Java profiler to identify the computationally intensive loop and the data structure it is iterating over.
(2) refactor the data structure into our parallel collection.
(3) refactor all loops over the data structure to use operators instead of **for**. The computationally intensive loop is refactored to run in parallel, while the rest are transformed to anonymous-class-operator form.

As with the already-parallel application, we limited the changes to what was necessary to fit the computation to the parallel collection API. We did not fix any races introduced by the parallelization. We only ensured the semantic equivalence by running the refactored versions on the same data sets and checking the outputs - we did not formally prove the modulo-data-races equivalence.

5.2. Measuring IteRace's performance

We first analyze each application using ITERACE with all the specialization techniques activated. We inspect each generated race warning in order to determine its root fault. Each *race warning* can be seen as a *possible error*. Typically, one *fault* can lead to multiple *errors*. In our case, one *fault* may lead to multiple *warnings*. If we cannot find a fault for a particular warning, we deem it as false.

For each of the applications in the test suite, we first run the analysis without any classes (in addition to JDK) marked with *is-library* (see Section 3.3, which discusses the *Bubble-up* technique). If there are no races or the faults generating the races are clear, we do not take

any further step. Still, for some of the applications, despite our techniques reducing the number of warnings by orders of magnitude, we still found ourselves needing to analyze tens to hundreds of warnings. Many of the warnings were over ten levels deep in the call graph, counting from the parallel loop. Figuring out whether the racing accesses are actually reachable during an actual execution, let alone whether truly shared objects can reach them, proved very challenging.

The solution came from using our *Bubble-up* technique in the iterative approach described in Section 3.3. In our experiments, it took up to 10 analysis reruns in order to find the set of library markings that best describe the fault. Cumulatively over all benchmarks, we needed just under 50 library markings given as simple regular expressions (see Section 4.3). For each application, it took us between a few minutes (for Em3D, MC, and JUnit) and a few hours (for Coref and Lucene) to reach the level where we fully understood all the reports. Most of the time was spent understanding the code so that we can answer the above question. We are not experts in the applications we analyzed, so we expect this effort to be lower for developers more familiar with the code. The results presented in the paper reflect this optimal balance.

Finally, we also analyze all applications with selectively deactivating various techniques to reveal their effect upon the analysis as a whole. When a technique is deactivated, its associated pointer analysis context sensitivity customizations are also deactivated. In addition to the three main techniques (*2-Threads*, *Filtering*, and *Bubble-up*), we also measure the effect of filtering warnings that come from correctly synchronized code, both at deep and at application level (see Section 3.5). Thus, there are five distinct parts of the analysis that can be turned on and off, hence 32 possible configurations. We run the analysis in all 32 configurations over all the applications. For each run, we measure runtime and number of warnings.

The machine running the experiments is a quad-core Intel Core i7 at 2.6 GHz (3720QM) with 16 GB of RAM. The JVM is allocated 4 GB of RAM. We implemented the race-detection techniques in Scala and we use the static analysis framework WALA, which is implemented in Java.

5.3. Comparison with JChord

We also analyze all projects using JChord. We have asked Mayur Naik, JChord's lead developer, for advice on how to best configure the tool. Accordingly, we configure JChord such that:

— it also reports races between instructions belonging to the same thread. By default, JChord only reports races between distinct abstract threads. As it models the threads executing a parallel loop as one abstract thread, the default behavior would ignore all races in parallel loops. Additionally, we have implemented a small tool that filters JChord's reports to remove races between the abstract thread representing the parallel loop and main thread. Such warnings are obviously false and are easy to filter out, so we considered it fair towards JChord to disregard them.
— it ignores races in constructor code. This reduces significantly the number of false positives reported by JChord but adds a source of unsoundness. While rare, constructors can have races, e.g., a constructor reads an object's field while another thread writes it. IteRace does not ignore races in constructors.
— it does not use conditional-must-not-alias analysis [Naik and Aiken 2007] as it is not currently available.

Additionally, we set JChord to ignore classes that IteRace models as *transitively-thread-safe* and not *circulates-unsafe-objects*. This increases the tool's precision without hampering safety.

Table II: **Overall results.** For JChord, the "#" column shows the number of warnings. "full" is the number of warnings reported by IteRace when all techniques are activated. "*Bubble-up*" has all techniques except *Bubble-up* activated. The analysis time for IteRace is with all techniques activated. "real" is how many of the warnings are real races (out of "full" for IteRace). Multiple warnings may be caused by the same program "fault". A warning may be false or benign, thus mapping to no fault. For MC, there is a real but benign race: multiple threads write the same constant to the same field in parallel.

| project | JChord | | | IteRace (our tool) | | | | |
| | t(s) | warnings | | t(s) | warnings | | | faults |
		#	real		*Bubble-up*	full	real	
Em3D	20	15	0	4.0	0	0	0	0
MC	22	44	1	6.5	1	1	1	0
JUnit	24	123	0	9.6	0	0	0	0
Coref	85	19.5k	-	101.0	138	32	24	2
Lucene	95	53.4k	-	122.9	775	19	2	2
WEKA	156	19.6k	-	155.9	400	1	1	1
CIlib	271	21.4k	-	84.0	179	2	2	1

JChord gives a very high number of warnings with their accesses deep in the call graph. We attempted to also inspect whether some of the warnings are true but it proved very difficult. As it was originally the case with IteRace, it is very hard to determine if a race reported deep in the application or library code is true. In the end, we could only complete the inspection for three of the case studies.

6. EVALUATION RESULTS

We first present our experience analyzing the evaluation suite applications using IteRace. Afterwards, we dig deeper and examine how effective is each of the techniques individually and in combination with others.

Table II shows an overview of the results. For context, the first three data columns show JChord's performance analyzing the evaluation suite applications. The next columns show IteRace's performance over the same applications.

A static race detection tool's runtime and results are heavily dependent on the underlying pointer analysis. Since JChord and IteRace have different underlying pointer analyses and abstraction choices, for the same application, their results differ both in terms of time and number of warnings. Still, JChord's results can give an idea about the effectiveness of a tool not implementing our techniques.

In terms of analysis time, our tool is faster for five out of the seven applications. JChord is slightly faster for two of the larger applications. On the one hand, JChord pays a penalty for its implementation communicating between various stages of the analysis through text files. On the other hand, IteRace's custom context sensitivity makes its pointer analysis slightly more expensive than JChord's context insensitive approach. Furthermore, our IFDS lockset algorithm is more expensive than JChord's graph traversal. Overall, the analyses times of both tools is low enough to be practical.

In terms of warnings, JChord reports an overwhelming number of warnings for five out of the seven applications (the table shows the number of warnings after the post-processing described in Section 5.3). For Em3D and JUnit the number of warnings is low enough to be inspected but all of the warnings are false.

For IteRace, we report here the number of warnings under two distinct configurations: with all the techniques activated ("full"), and with all techniques but *Bubble-up* activated ("no-*Bubble-up*"). We first discuss IteRace's results with all techniques activated.

For two applications, our tool doesn't report any races, correctly deeming them safe. For five out of the seven applications IteRace only identified the correct races, i.e. it did not report any false warnings. In our experience, after *Bubble-up*, the number of warnings is very low, and the reported accesses are close to the parallel loop body and they are a good indicator of the underlying fault. The *Bubble-up* technique requires some programmer effort in gradually removing library markings (see Section 5.2). Thus, for a direct comparison with JChord, we also show, under the "no-*Bubble-up*" column the number or races with *Bubble-up* deactivated. Deactivating *Bubble-up* is the same as having no classes (not even JDK) marked as library code, thus requiring the same amount of effort as JChord.

Let us look at the issue of missed races. IteRace's underlying approach is very similar to JChord's. *Synchronized* is the application-level version of the same may-alias lockset-based filtering used in JChord. *2-Threads* and *Bubble-up* are inherently safe and *Filtering* is safe when used correctly (see Section 4). Thus, it is highly unlikely that IteRace will miss any true races JChord finds.

Furthermore, at first glance, the number of warnings may seem large (32 and 19 in the case of Coref and Lucene, respectively). Still, the way IteRace reports them makes them easy to understand. As shown in Section 2, in IteRace's standard output the races are not reported as pairs but as race sets on fields of abstract objects. A race set on one field of an object is shown as a set of α accesses and a set of β accesses - races are obtained by cross-product. E.g., one single race set of 4 write (α) accesses and 5 β accesses would generate 20 race warnings, as counted in Table II. Still, it is relatively easy for a programmer familiar with the application to inspect 4+5 accesses involving the same field of the same object.

6.1. Case studies

Em3D and JUnit are race free and IteRace correctly reports no warnings for any of them. MC contains a benign race where a static global is initialized with the same value in every iteration. This is a true race but cannot be considered a fault. We have not accounted for this type of scenario so our tool issues a warning. JChord found this race, also.

Coref is one of the applications that we parallelized ourselves and we contributed back the parallel version. The developers of the project told us that there is no interaction between the iterations of the parallel loop. IteRace reports 32 warnings out of which 24 are true. The false warnings were due to WALA's imprecise modeling of Java object cloning. Most of the warnings are rooted in the sharing introduced via two static fields used for caching purposes. The developers confirmed the faults and fixed the application by making the static fields thread-local.

For Lucene, IteRace reports 19 warnings out of which 2 are true. First, there is an unsynchronized access to a custom, thread-unsafe, String interning class. Second, there is an unsynchronized access to a factory method of the DateFormat class. The access leads to an atomicity violation in the JDK LocalServiceProviderPool class. We reported the problem to the JDK developers. The problem is mostly benign assuming correct implementation of other classes. Still, it had already been fixed in the latest JDK release.

For WEKA, the analysis hits the right target with great precision. While running the analysis at a deeper level also yields false positives, after *Bubble-up*, the analysis only makes one warning report, a correct one: all loop iterations share the same thread-unsafe custom collection object.

For CIlib, the analysis is again very precise, reporting only two warnings, both true. We reported them to CIlib developers and they confirmed and fixed the fault [CIl 2015].

Table III: **Runtime under various configurations.** (seconds)
T - *2-Threads*, F - *Filtering*, B - *Bubble-up*, S - *Synchronized*

T	F	B	S	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib	**avg.**
				3.9	5.1	6.7	34.1	23.8	56.9	29.6	**15.2**
			●	4.0	6.2	8.7	67.0	140.5	151.1	49.5	**28.6**
		●		4.0	6.0	7.8	354.3	428.1	131.5	54.1	**41.3**
		●	●	4.0	6.7	9.2	536.7	571.9	163.4	62.3	**50.1**
	●			4.0	5.0	6.7	36.5	27.5	3307.5	29.2	**27.9**
	●		●	4.0	5.4	7.5	47.1	62.1	6448.7	36.4	**38.0**
	●	●		4.0	5.1	6.8	38.9	31.5	60.9	30.6	**16.4**
	●	●	●	4.1	5.5	7.5	43.9	51.0	74.8	34.6	**19.3**
●				4.1	5.2	7.7	74.9	42.0	92.7	68.4	**22.9**
●			●	4.1	6.5	9.6	124.2	255.2	334.0	111.7	**43.7**
●		●		4.0	5.4	8.0	80.8	46.3	101.9	72.4	**24.2**
●		●	●	4.0	6.6	9.6	116.7	146.5	232.9	101.1	**37.4**
●	●			4.0	5.3	7.7	90.0	49.0	109.6	72.4	**24.8**
●	●		●	4.1	6.3	9.8	102.8	142.9	157.7	83.5	**33.8**
●	●	●		4.2	5.3	7.7	87.9	49.7	102.1	67.7	**24.4**
●	●	●	●	4.0	6.5	9.6	101.0	122.9	155.9	84.0	**32.8**

In a previous version of ITERACE [Radoi and Dig 2013], precision was dropping significantly when aiming the analysis at some parts of CIlib's extensive algorithm library. It was raising many false warnings along with the aforementioned true ones. We traced many of the false warnings to a source of imprecision in WALA's pointer analysis method call abstraction: WALA propagates all actual parameter objects to the formal parameters of all target call graph nodes, regardless of object context sensitivity. This made the technique described at the end of Section 3.1 less effective when the actual parameter points to both shared and non-shared objects. We have since improved WALA's pointer analysis engine to overcome this issue and configured ITERACE's context sensitivity to make better use of the improved precision. This resulted in a significant drop in the number of false positive warnings for Lucene and CIlib (from 97 to 19, and from 735 to 2, respectively[1]).

Generally, we found that ITERACE performs well on new programs. Still, we do expect that the tool's performance would degrade for programs that make heavy use of reflection – none of the programs in our evaluation did. In terms of programer effort, each new case study did not require many new tread-safety markings. On the contrary, we were able to reuse many of the specifications written previously. We just needed to add one marking for WEKA and two markings for each of CIlib and Coref. One of the CIlib markings was for the Vector class in Guava, a third-party library.

6.2. Effect of each specialization technique

Tables III shows the runtime and Table V shows the number of warnings reported by our analysis under 16 of the 32 possible configurations. We are not showing results for filtering warnings based on deep synchronization for all configurations due to its limited impact –

---

[1]The "from" numbers are from experiments ran after compiling ITERACE with the latest WALA version. They differ from our previous work. This is likely due to changes to WALA that occurred in the meantime.

Table IV: **Runtime breakdown for the best configuration.** (seconds)

Stage	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib
pointer analysis	3.8	5.1	7.3	85.1	46.5	102.7	67.3
potential races	0.1	0.2	0.4	3.6	2.8	3.4	3.2
locksets algorithm	0.0	1.0	1.8	11.8	71.8	49.1	13.3
other	0.1	0.2	0.1	0.5	1.2	0.7	0.2
Total	4.0	6.5	9.6	101.0	122.9	155.9	84.0

Table V: **Number of warnings under various configurations.**
(racing pairs of accesses)
T - *2-Threads*, F - *Filtering*, B - *Bubble-up*, S - *Synchronized*

T	F	B	S	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib
				1	2477	2370	66K	135K	50K	67K
			•	1	2471	2334	66K	133K	50K	39K
		•		1	747	222	516K	228K	6148	9686
		•	•	1	747	207	516K	226K	6143	9637
	•			1	179	48	15K	28K	6647	8120
	•		•	1	147	20	15K	26K	6515	8115
	•	•		1	155	36	473	8278	1344	2048
	•	•	•	1	155	30	473	6391	1267	2039
•				0	53	87	14K	15K	5450	21K
•			•	0	53	70	14K	13K	5377	3104
•		•		0	3	3	18K	2426	139	518
•		•	•	0	3	0	18K	538	139	476
•	•			0	1	17	138	2099	415	183
•	•		•	0	1	0	138	775	400	179
•	•	•		0	1	3	32	1906	1	8
•	•	•	•	0	1	0	32	19	1	2

see the end of the section. Each row shows the results for one configuration – a dot denotes an activated technique.

The best results, i.e., the lowest number of warnings, are obtained when all techniques are activated (last row of Table V).

In the best configuration, ITERACE finishes the analysis in under three minutes for all applications and in around half a minute on average. Table IV shows a breakdown of the time spent in different stages of the analysis for the best configuration (last row in Table III). The underlying pointer analysis dominates the running time, followed by the IFDS lockset allocation algorithm (Sec. 3.4). Creating the potential races set is fast thanks to the hierarchical data structure described in Sec. 4.4. The other stages have a negligible effect.

Tables VI, VII, VIII, and IX highlight the effect of activating/deactivating each technique on the number or warnings. These tables are derived from Table V. The value in each cell is the ratio between the number of races on a certain configuration with the technique deactivated and the number of races with the technique activated. For example, the value in cell at the intersection of the next to last row (*Filtering* and *Bubble-up* activated, *Synchronized* de-

Table VI: **Effect of *2-Threads* on the number of warnings.** (improvement ratio, see third paragraph of Sec. 6.2)

F	B	S	Em3D	mc	JUnit	Coref	Lucene	WEKA	CIlib
			∞	46.74	27.24	4.44	8.68	9.30	3.11
		•	∞	46.62	33.34	4.45	9.84	9.41	12.76
	•		∞	249.00	74.00	27.38	94.28	44.23	18.70
	•	•	∞	249.00	∞	27.38	421.63	44.19	20.25
•			∞	179.00	2.82	111.68	13.51	16.02	44.37
•		•	∞	147.00	∞	110.72	34.70	16.29	45.34
•	•		∞	155.00	12.00	14.78	4.34	1344.00	256.00
•	•	•	∞	155.00	∞	14.78	336.37	1267.00	1019.50

Table VII: **Effect of *Filtering* on the number of warnings.** (improvement ratio, see third paragraph of Sec. 6.2)

T	B	S	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib
			1.00	13.84	49.38	4.30	4.79	7.63	8.28
		•	1.00	16.81	116.70	4.34	4.97	7.77	4.88
	•		1.00	4.82	6.17	1092.23	27.63	4.57	4.73
	•	•	1.00	4.82	6.90	1092.23	35.49	4.85	4.73
•			NaN	53.00	5.12	108.12	7.45	13.13	117.99
•		•	NaN	53.00	∞	107.85	17.51	13.44	17.34
•	•		NaN	3.00	1.00	589.75	1.27	139.00	64.75
•	•	•	NaN	3.00	NaN	589.75	28.32	139.00	238.00

activated) and the "JUnit" column in Table VI is obtained from Table V, column "JUnit", by dividing the cell in row 7 (*2-Threads* deactivated, *Filtering* and *Bubble-up* activated, *Synchronized* deactivated) by the cell in the next to last row (*2-Threads* how activated, *Filtering* and *Bubble-up* activated, *Synchronized* deactivated). A higher ratio means the activated technique filters out more warnings, which is an improvement. ∞ denotes a situation where the number of warnings is reduced to 0. 1.0 means no improvement. NaN denotes a situation where the number of warnings was 0 with the technique deactivated and it remains 0. A subunitary value means that the number of warnings has increased.

Table VI shows that *2-Threads* (modeling each loop with two distinct threads) significantly improves the results independent of other techniques. Upon inspection we found that, as expected, the filtered out warnings are on objects that are thread-local by being either created and not escaped from the current iteration or unique to each element of the collection. In the case of Em3D, activating *2-Threads* correctly removed all warnings, independent of the other techniques.

Table VII shows that *Filtering* has a powerful effect for all larger applications. Deactivating the *Filtering* phase is equivalent to not giving any thread-safety specifications. The filtered out warnings mostly involve accesses to library classes, e.g., synchronized I/O, Java security, regex, and concurrent or synchronized collections.

Table VIII shows the effect of *Bubble-up*. Its main value is not in reducing the number of warnings but in making them more programmer friendly. As the technique maps deep warnings into application-level warnings, and, as it is common for one library class to be used repeatedly throughout the application, *Bubble-up* may inflate the number of warnings.

Table VIII: **Effect of *Bubble-up* on the number of warnings.** (improvement ratio, see third paragraph of Sec. 6.2)

T	F	S	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib
			1.00	3.32	10.68	0.13	0.59	8.25	6.94
		●	1.00	3.31	11.28	0.13	0.59	8.24	4.11
	●		1.00	1.15	1.33	32.58	3.43	4.95	3.96
	●	●	1.00	0.95	0.67	32.30	4.21	5.14	3.98
●			NaN	17.67	29.00	0.79	6.45	39.21	41.68
●		●	NaN	17.67	∞	0.79	25.23	38.68	6.52
●	●		NaN	1.00	5.67	4.31	1.10	415.00	22.88
●	●	●	NaN	1.00	NaN	4.31	40.79	400.00	89.50

Table IX: **Effect of *Synchronized* on the number of race warnings.** (improvement ratio, similar to Table 5).

T	F	B	Em3D	MC	JUnit	Coref	Lucene	WEKA	CIlib
			1.00	1.00	1.02	1.00	1.02	1.00	1.70
		●	1.00	1.00	1.07	1.00	1.01	1.00	1.01
	●		1.00	1.22	2.40	1.01	1.05	1.02	1.00
	●	●	1.00	1.00	1.20	1.00	1.30	1.06	1.00
●			NaN	1.00	1.24	1.00	1.15	1.01	6.96
●		●	NaN	1.00	∞	1.00	4.51	1.00	1.09
●	●		NaN	1.00	∞	1.00	2.71	1.04	1.02
●	●	●	NaN	1.00	∞	1.00	100.32	1.00	4.00

This effect is revealed by the sub-unitary values in rows 1, 2, 4, 5, and 6. Still, when combined with *Filtering* (rows 3, 4, 7, and 8) the negative effect is reversed and we see improvement in most cases. This is because most extra warnings came from correctly-synchronized library classes.

Table IX shows that, surprisingly, the lockset-based static filtering, i.e., *Synchronized*, does little to improve analysis results for larger projects, even in the absence of *Filtering*. The only project where *Synchronized* has a very significant impact is Lucene. Also, for JUnit, *Synchronized* removed 3 false warnings, bringing the number of reports to 0. For CIlib, *Synchronized* eliminated 6 warnings, leaving behind only the 2 true warnings.

7. RELATED WORK

7.1. Dynamic analyses

Dynamic race detectors have been the favored approach in the last decade. Their main advantage over static approaches is the significantly lower number of false warnings. This advantage is counterbalanced by dynamic analyses' failure to catch races that are not "close" to the analyzed execution and the high runtime cost of the more precise tools. A common approach is to compute some form of order relation, e.g. happens-before, over the events of an observed execution trace and, based on these relations, infer race conditions [Christiaens and De Bosschere 2001; Ronsse and De Bosschere 1999; Adve et al. 1991; Choi et al. 1991; Mellor-Crummey 1991; Dinning and Schonberg 1990; Schonberg 1989; Smaragdakis et al. 2012]. This approach can miss many races so lockset-based race detectors have been devel-

oped as an alternative that catches more races at the expense of false positives [Nishiyama 2004; von Praun and Gross 2001; Savage et al. 1997; Choi et al. 2002]. There are also hybrid approaches that combine both techniques [Yu et al. 2005; Pozniansky and Schuster 2007; Chen et al. 2008; Flanagan and Freund 2009].

Notable in the context of our specialized race detector, Raman et al.[Raman et al. 2010; 2012] take advantage of the structured (async) parallelism in the analyzed program to significantly reduce the overhead introduced by dynamic race detection. They use a representation of the dynamic thread structure to get a more compact memory access history. Like us, they found that specialization improves the effectiveness of previous techniques.

Similar to dynamic race detectors, static race detectors also vary between higher precision, lower scalability [Henzinger et al. 2004; Naik and Aiken 2007] and lower precision, better scalability [Pratikakis et al. 2006; Naik et al. 2006; Voung et al. 2007; Kahlon et al. 2009; Pratikakis et al. 2011]. Also, annotations can be used to improve the performance of the analysis [Abadi et al. 2006].

7.2. Static analyses for C and other languages

Several race analyses have been proposed for C or variants [Engler and Ashcraft 2003; Grossman 2003; Qadeer and Wu 2004]. Henzinger et al. [Henzinger et al. 2004] present a model checking approach that is both path and flow sensitive, and models thread contexts. Pratikakis et al. present LOCKSMITH [Pratikakis et al. 2006; 2011], a type-based analysis that computes context-senstitive *correlations* between lock and memory accesses. RELAY [Voung et al. 2007] proposes a slightly less precise but more scalable analysis that summarizes the effects of functions using *relative locksets*. Although they are now applied to C programs, both of these techniques could be adapted to improve the precision of Java analyses, including ours.

The polyhedral model can be used for detecting races [Feautrier 1991; Yuki et al. 2013; Basupalli et al. 2011]. Still, so far, the polyhedral approach has been limited to array-manipulating programs with restricted control-flow (e.g., there is no support for objects or recursive functions).

7.3. Static analyses for Java

Flanagan et al. [Flanagan and Freund 2000] proposed using type checking systems to find races. Boyapati et al. [Boyapati and Rinard 2001; Boyapati et al. 2002] introduced the concept of *ownership* to improve the results. Type-based systems perform very well but they require a significant amount of annotation from the programmer. Different approaches have been proposed to automatically infer the annotations [Flanagan and Freund 2001; Agarwal and Stoller 2004; Rose et al. 2005; Flanagan and Freund 2007].

Praun et al. [von Praun and Gross 2003] propose an Object Use Graph model that statically approximates the happens-before relation between accesses to a specific object.

Choi et al. [Choi et al. 2001] proposes a thread-sensitive but context-insensitive race detector. They use the strongly connected components of an inter-procedural thread-sensitive control flow graph to compute must-alias relations between locks and threads. Using this, they find a limited number of definite races. ITERACE uses the idea of thread-sensitivity but specializes the modeling of the parallel loops, significantly increasing precision.

Naik et al. [Naik et al. 2006] builds an object-sensitive analysis that uses thread-escape to lower the false positive rate. In a subsequent article [Naik and Aiken 2007], they present a conditional-must-not-alias analysis for solving aliasing relationships between locks.

8. CONCLUSION

By specializing static data race detection, we can make it practical. This paper presents three techniques, implemented in a tool ITERACE, that is specialized to the new parallel features for collections that have been introduced in Java 8. The restricted thread structure

of parallel loops combined with loop operations expressed as lambda expressions allows for better precision in the heap modeling while maintaining scalability.

Our evaluation shows that the tool implementing this approach is fast, does not hinder the programmer with many warnings, and it finds new bugs that were confirmed and fixed by the developers. Thus, ITERACE can also be used in scenarios with high interactivity, e.g., refactoring for parallelism [Dig et al. 2009b; Gyori et al. 2013; Dig et al. 2009a], that require fast and precise analyses.

ACKNOWLEDGMENTS

REFERENCES

2015. CIlib bug. https://github.com/CIlib/CIlib/issues/111. (2015).

2015. Concurrency JSR-166 Interest Site - ParallelArray. http://g.oswego.edu/dl/concurrency-interest/. (2015).

2015. JDK8. http://jdk8.java.net. (2015).

2015. Microsoft TPL. http://msdn.microsoft.com/en-us/library/dd460717.aspx. (2015).

2015. State of the Lambda: Libraries Edition. http://openjdk.java.net/projects/lambda/. (2015).

2015. Threading Building Blocks. http://threadingbuildingblocks.org/. (2015).

2015. WALA documentation. http://wala.sourceforge.net/. (2015).

2015. YourKit Java Profiler. http://www.yourkit.com. (2015).

Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for safe locking: Static race detection for Java. *TOPLAS* 28 (March 2006), 207–255. Issue 2. DOI:http://dx.doi.org/10.1145/1119479.1119480

Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. 1991. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News* 19 (April 1991), 234–243. Issue 3. DOI:http://dx.doi.org/10.1145/115953.115976

Rahul Agarwal and Scott Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *VMCAI*, Bernhard Steffen and Giorgio Levi (Eds.). Lecture Notes in Computer Science, Vol. 2937. Springer Berlin / Heidelberg, 77–108.

Vamshi Basupalli, Tomofumi Yuki, Sanjay Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and David Wonnacott. 2011. ompVerify: polyhedral analysis for the OpenMP programmer. In *OpenMP in the Petascale Era*. Springer, 37–53.

E. Bengtson and D. Roth. 2008. Understanding the value of features for Coreference resolution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 294–303.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*. ACM Press, New York, NY, USA, 169–190. DOI:http://dx.doi.org/10.1145/1167473.1167488

Eric Bodden and Klaus Havelund. 2008. Racer: effective race detection using AspectJ. In *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08)*. ACM, New York, NY, USA, 155–166. DOI:http://dx.doi.org/10.1145/1390630.1390650

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. 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 (OOPSLA '02)*. ACM, New York, NY, USA, 211–230. DOI:http://dx.doi.org/10.1145/582419.582440

Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01)*. ACM, New York, NY, USA, 56–69. DOI:http://dx.doi.org/10.1145/504282.504287

J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. 1999. A Benchmark Suite for High Performance Java. In *Java Grande*. ACM Press, 81–88.

B. Cahoon and K.S. McKinley. 2001. Data flow analysis for software prefetching linked data structures in Java. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. 280 –291. DOI:http://dx.doi.org/10.1109/PACT.2001.953309

Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. 2008. jPredictor: a predictive runtime analysis tool for Java. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 221–230. DOI:http://dx.doi.org/doi:10.1145/1368088.1368119

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI '02)*. ACM, New York, NY, USA, 258–269. DOI:http://dx.doi.org/10.1145/512529.512560

Jong-Deok Choi, Alexey Loginov, and Vivek Sarkar. 2001. *Static Datarace Analysis for Multithreaded Object-Oriented Programs*. Technical Report. IBM Research Division, Thomas J. Watson Research Centre.

Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. 1991. Techniques for debugging parallel programs with flowback analysis. *TOPLAS* 13 (October 1991), 491–530. Issue 4. DOI:http://dx.doi.org/10.1145/115372.115324

Mark Christiaens and Koen De Bosschere. 2001. TRaDe, a topological approach to on-the-fly race detection in Java programs. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267847.1267862

Danny Dig, John Marrero, and Michael D. Ernst. 2009a. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 397–407. DOI:http://dx.doi.org/10.1109/ICSE.2009.5070539

Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. 2009b. Relooper: refactoring for loop parallelism in Java. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09)*. ACM, New York, NY, USA, 793–794. DOI:http://dx.doi.org/10.1145/1639950.1640018

A. Dinning and E. Schonberg. 1990. An empirical comparison of monitoring algorithms for access anomaly detection. *SIGPLAN Not.* 25 (February 1990), 1–10. Issue 3. DOI:http://dx.doi.org/10.1145/99164.99165

Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.* 37 (October 2003), 237–252. Issue 5. DOI:http://dx.doi.org/10.1145/1165389.945468

Paul Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming* 20 (1991).

Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00)*. ACM, New York, NY, USA, 219–232. DOI:http://dx.doi.org/10.1145/349299.349328

Cormac Flanagan and Stephen N. Freund. 2001. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*. ACM, New York, NY, USA, 90–96. DOI:http://dx.doi.org/10.1145/379605.379687

Cormac Flanagan and Stephen N. Freund. 2007. Type inference against races. *Sci. Comput. Program.* 64 (January 2007), 140–165. Issue 1.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. DOI:http://dx.doi.org/10.1145/1542476.1542490

Dan Grossman. 2003. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation (TLDI '03)*. ACM, New York, NY, USA, 13–25. DOI:http://dx.doi.org/10.1145/604174.604177

Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the Gap from Imperative to Functional Programming Through Refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 543–553. DOI:http://dx.doi.org/10.1145/2491411.2491461

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. DOI:http://dx.doi.org/10.1145/1656274.1656278

Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. 2007. Component-Based Lock Allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, Washington, DC, USA, 353–364. DOI:http://dx.doi.org/10.1109/PACT.2007.23

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*. ACM, New York, NY, USA, 1–13. DOI:http://dx.doi.org/10.1145/996841.996844

Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61. DOI:http://dx.doi.org/10.1145/379605.379665

Ranjit Jhala and Rupak Majumdar. 2007. Interprocedural analysis of asynchronous programs. *SIGPLAN Not.* 42 (January 2007), 339–350. Issue 1. DOI:http://dx.doi.org/10.1145/1190215.1190266

Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. 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 on European software engineering conference and foundations of software engineering symposium (ESEC/FSE '09)*. ACM, New York, NY, USA, 13–22. DOI:http://dx.doi.org/10.1145/1595696.1595701

Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. 2010. A dynamic evaluation of the precision of static heap abstractions. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*. ACM, New York, NY, USA, 411–427. DOI:http://dx.doi.org/10.1145/1869459.1869494

Yu Lin and Danny Dig. 2013. CHECK-THEN-ACT Misuse of Java Concurrent Collections. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, Washington, DC, USA, 164–173. DOI:http://dx.doi.org/10.1109/ICST.2013.41

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*. ACM, New York, NY, USA, 134–143. DOI:http://dx.doi.org/10.1145/1542476.1542491

John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (ICS '91)*. ACM, New York, NY, USA, 24–33. DOI:http://dx.doi.org/10.1145/125826.125861

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14 (January 2005), 1–41. Issue 1. DOI:http://dx.doi.org/10.1145/1044834.1044835

Mayur Naik and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07)*. ACM, New York, NY, USA, 327–338. DOI:http://dx.doi.org/10.1145/1190216.1190265

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319. DOI:http://dx.doi.org/10.1145/1133981.1134018

Mayur Naik, Percy Liang, and Mooly Sagiv. 2010. Static Thread-Escape Analysis vis Dynamic Heap Abstractions. (2010). from Naik's website.

Hiroyasu Nishiyama. 2004. Detecting data races using dynamic escape analysis based on read barrier. In *VM*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1267242.1267252

R. O'Callahan and J.D. Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*, Vol. 38. ACM, 167–178.

Semih Okur and Danny Dig. 2012. How Do Developers Use Parallel Libraries?. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 54, 11 pages. DOI:http://dx.doi.org/10.1145/2393596.2393660

Eli Pozniansky and Assaf Schuster. 2007. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340. DOI:http://dx.doi.org/10.1002/cpe.1064

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI '06)*. ACM, New York, NY, USA, 320–331. DOI:http://dx.doi.org/10.1145/1133981.1134019

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* 33, Article 3 (January 2011), 55 pages. Issue 1. DOI:http://dx.doi.org/10.1145/1889997.1890000

Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*. ACM, New York, NY, USA, 14–24. DOI:http://dx.doi.org/10.1145/996841.996845

Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. 2009. Multicoresdk: a practical and efficient data race detector for real-world applications. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM, 5.

Cosmin Radoi and Danny Dig. 2013. Practical Static Race Detection for Java Parallel Loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 178–190. DOI:http://dx.doi.org/10.1145/2483760.2483765

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-finish Parallelism. In *Proceedings of the First International Conference on Runtime Verification (RV'10)*. Springer-Verlag, Berlin, Heidelberg, 368–383. http://dl.acm.org/citation.cfm?id=1939399.1939430

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 531–542. DOI:http://dx.doi.org/10.1145/2254064.2254127

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95)*. ACM, New York, NY, USA, 49–61. DOI:http://dx.doi.org/10.1145/199448.199462

Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.* 17 (May 1999), 133–152. Issue 2. DOI:http://dx.doi.org/10.1145/312203.312214

James Rose, Nikhil Swamy, and Michael Hicks. 2005. Dynamic inference of polymorphic lock types. *Science of Computer Programming* 58, 3 (2005), 366 – 383.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15 (November 1997), 391–411. Issue 4. DOI:http://dx.doi.org/10.1145/265924.265927

D. Schonberg. 1989. On-the-fly detection of access anomalies. *SIGPLAN Not.* 24 (June 1989), 285–297. Issue 7. DOI:http://dx.doi.org/10.1145/74818.74844

M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.* (1981).

Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2011. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 401–410. DOI:http://dx.doi.org/10.1145/1985793.1985848

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '12)*. ACM, New York, NY, USA, 387–400. DOI:http://dx.doi.org/10.1145/2103656.2103702

Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. 2011. Are Java Programmers Transitioning to Multicore?: A Large Scale Study of Java FLOSS. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops)*. ACM, New York, NY, USA, 123–128. DOI:http://dx.doi.org/10.1145/2095050.2095072

Christoph von Praun and Thomas R. Gross. 2001. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01)*. ACM, New York, NY, USA, 70–82. DOI:http://dx.doi.org/10.1145/504282.504288

Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming language design and implementation (PLDI '03)*. 115–128.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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 (ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214. DOI:http://dx.doi.org/10.1145/1287624.1287654

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.* 39 (October 2005), 221–234. Issue 5. DOI:http://dx.doi.org/10.1145/1095809.1095832

Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13)*. ACM, New York, NY, USA, 23–34. DOI:http://dx.doi.org/10.1145/2442516.2442520