PRACTICAL STATIC RACE DETECTION FOR JAVA PARALLEL LOOPS

BY

COSMIN A. RĂDOI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Advisor:

Adjunct Assistant Professor Danny Dig

Abstract

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 to be 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 few race warnings, thus avoiding a common pitfall of static analyses. The tool 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.

Acknowledgments

This work has benefited from the time and support of many people. I would like to thank my advisor, Danny Dig. He has trusted my research ability and has guided me with patience throughout the development of this project.

I thank Codruta Girlea, Stas Negara, Sandro Badame, Francesco Sorrentino, Rajesh Karmani, Nicholas Chen, Semih Okur, Samira Tasharofi, Milos Gligoric, Traian-Florin Şerbănuţă, Manu Sridharan, Darko Marinov, and Vikram Adve for their feedback. I also thank Mihai Codoban and Caius Brindescu for their help in making an unbiased evaluation.

This research is partly funded through NSF CCF-1213091 and CCF-1219027 grants, an Intel gift grant, and the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

Table of Contents

Chapter 1

Introduction

The recent prevalence of multi-core processors has increased the use of shared-memory parallel programming. Loop parallelism is often the first choice when attempting to speed up programs [48]. The major programming languages have parallel constructs or libraries that provide extensive support for loop parallelism, e.g., `Parallel.For` in .NET TPL [4], `.parallel()` in the upcoming Java 8 collections [5], `parallel_for` in C++ TBB [6]. Still, programs with parallel loops are subject to the major plague in 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.

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 [8, 13, 34, 35, 37–39, 43–45, 51, 64] 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. Static race detectors rarely miss races but are faced with the opposite problem: despite continuous improvements, they still report impractically-many false warnings. For example, we applied JChord [44], 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 [9, 21, 22, 25, 30, 39–41, 45–47, 55, 57, 58, 60, 62], 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 scala-

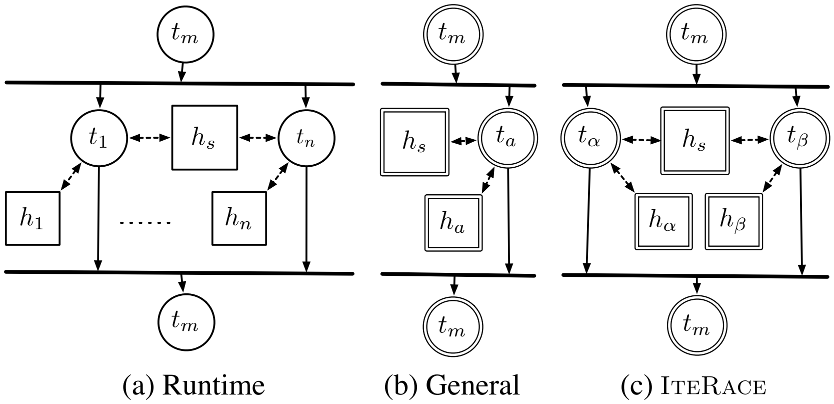


Figure 1.1: **Modeling a parallel loop.** Circles are threads, squares heap regions. Double line denotes abstraction.

bility. 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 races. We present three *specialization* techniques that contribute to these goals: (i) *2-Threads* – make the analysis aware of the threading and data-flow structure of loop-parallel operations, (ii) *Bubble-up* – report races in application code, not in libraries, and (iii) *Filtering* – filter the race warnings based on a thread-safety model of library classes. We implemented these techniques in a tool, IteRace, and empirically validated how well they work individually, and in tandem.

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.1.1.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 [44, 51] (see Fig.1.1.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. 1.1.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.

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 [36], we use this model to specialize the context sensitivity to increase precision and lower runtime.

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.

Contributions

This paper makes 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 [5], (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 [44]. The results show that our specialized approach is sufficiently fast and precise to be practical. It runs it at most a few minutes and reports very few warnings for many of the case studies.

  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 alone and in tandem with other techniques.

This thesis is a revised version of work previously published by the author [53].

Chapter 2

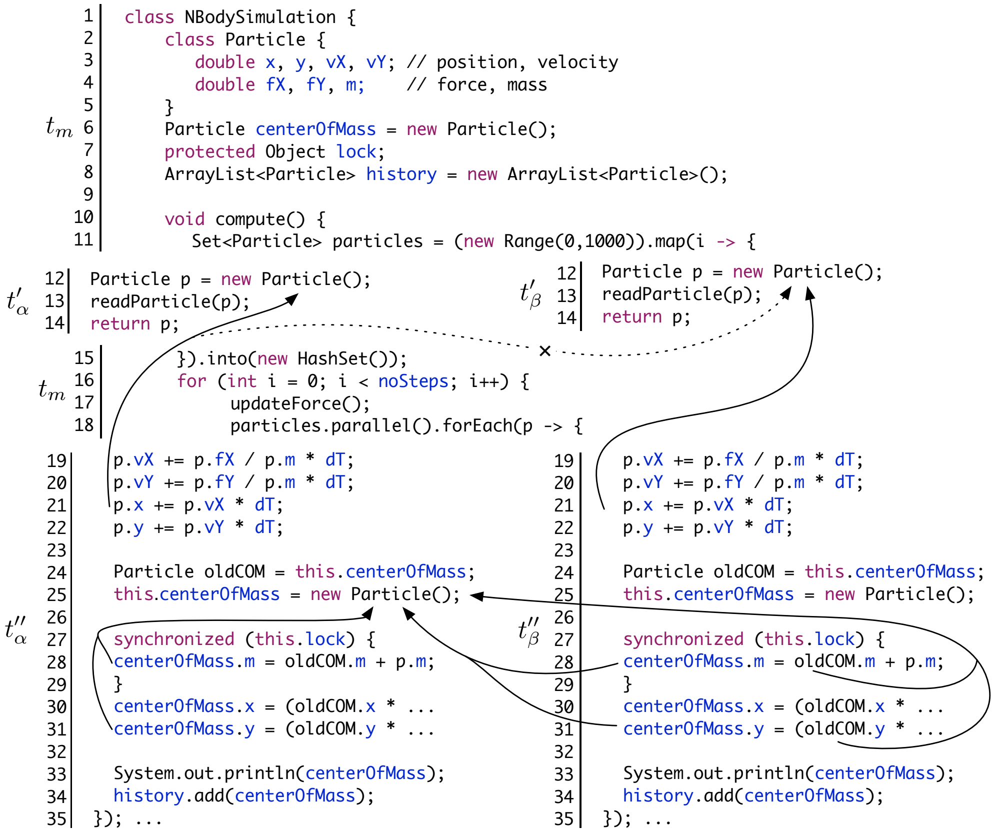
Motivating example

To illustrate our analysis, we use a simple N-body simulation implementation, shown partially in Fig. 2.1; 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 to be introduced in Java 8 [3]. In Java 8, clients can call the `parallel()` method on any `Collection` 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).

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 `NBodySimulation` (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 `NBodySimulation` 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



```
 1  class NBodySimulation {
 2      class Particle {
 3         double x, y, vX, vY; // position, velocity
 4         double fX, fY, m;    // force, mass
 5      }
 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();
13  readParticle(p);
14  return p;
```

```
12  Particle p = new Particle();
13  readParticle(p);
14  return p;
```

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

```
19    p.vX += p.fX / p.m * dT;
20    p.vY += p.fY / p.m * dT;
21    p.x += p.vX * dT;
22    p.y += p.vY * dT;
23
24    Particle oldCOM = this.centerOfMass;
25    this.centerOfMass = new Particle();
26
27    synchronized (this.lock) {
28    centerOfMass.m = oldCOM.m + p.m;
29    }
30    centerOfMass.x = (oldCOM.x * ...
31    centerOfMass.y = (oldCOM.y * ...
32
33    System.out.println(centerOfMass);
34    history.add(centerOfMass);
35  }); ...
```

```
19    p.vX += p.fX / p.m * dT;
20    p.vY += p.fY / p.m * dT;
21    p.x += p.vX * dT;
22    p.y += p.vY * dT;
23
24    Particle oldCOM = this.centerOfMass;
25    this.centerOfMass = new Particle();
26
27    synchronized (this.lock) {
28    centerOfMass.m = oldCOM.m + p.m;
29    }
30    centerOfMass.x = (oldCOM.x * ...
31    centerOfMass.y = (oldCOM.y * ...
32
33    System.out.println(centerOfMass);
34    history.add(centerOfMass);
35  }); ...
```

Figure 2.1: **Visual representation of how our analysis sees 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 missing in our model.

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 `NBodySimulation` 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`.

The next section explains how IteRace correctly identifies all the races described above. 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.

Chapter 3

Race detection

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

Figure 3.1 presents a high level overview of IteRace. WALA [7] provides the underlying Andersen-style static pointer analysis. The call graph is computed on-the-fly along with the heap model, based on context sensitivity. Each of our techniques specializes the context sensitivity, as detailed in sections 3.1, 3.3, and 3.2. 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 [54].

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. 3.4) 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.

3.1 2-Threads program model

The main thread of the program is modeled by an abstract thread t_m (lines 1-8 and 16-17 in our example). As outlined in Fig. 1.1, the concrete threads executing each loop are modeled by two abstract threads, t_α and t_β. In our example (Fig. 2.1), $\langle t'_\alpha, t'_\beta \rangle$ and $\langle t''_\alpha, t''_\beta \rangle$ model the threads executing the parallel

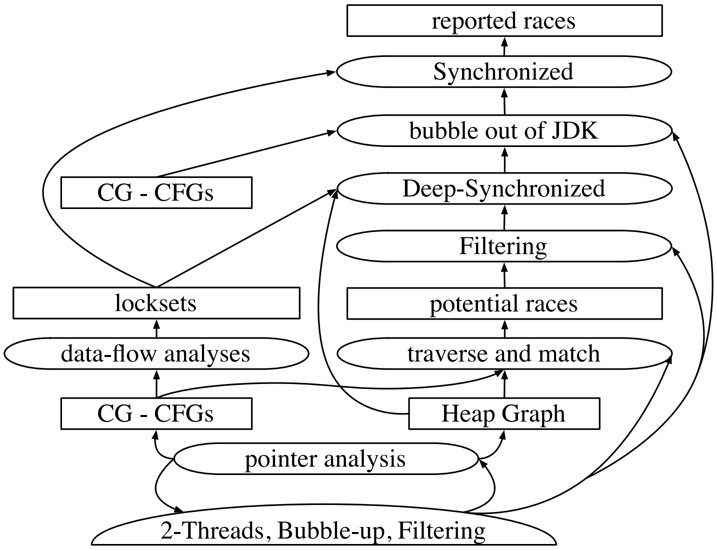


Figure 3.1: **Analysis overview.** Ovals represent different sub-analyses. Rectangles represent intermediate and final data structures. The bottom half-oval represents the specialized context sensitivity mechanism.

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.

The analysis matches loops operating on the same collection, e.g., $\langle t'_\alpha, t'_\beta \rangle$ and $\langle t''_\alpha, t''_\beta \rangle$, using may-alias. If the collection references do not alias in a concrete execution, the analysis may introduce spurious warnings, but it is still safe. Additionally, the technique dynamically adds levels of object sensitivity [42] in order to precisely track the collections of interest through the program.

The analysis maintains a special modeling for each collection of interest. The elements of a collection are modeled by two abstract fields, e_α and e_β. Fig. 3.2 shows how each of the abstract threads, t_α and t_β, processes one of the abstract fields, e_α respectively e_β. This modeling 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 respective t_β. Also, it sees that the result of 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 [5], 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 2.1, 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 with `p` in t'_α but different from `p` from t'_β.

$$\begin{array}{r|ll}
\text{c.forEach(op)} & \texttt{op}(e_\alpha) & [t_\alpha] \\
 & \texttt{op}(e_\beta) & [t_\beta] \\
\hline
 & e_\alpha = \texttt{op}(e_\alpha) & [t_\alpha] \\
\text{c.map(op)} & e_\beta = \texttt{op}(e_\beta) & [t_\beta] \\
 & \textit{return } c & [t_m] \\
\hline
 & x_1 = \texttt{op}(e_\alpha, \text{base}) & [t_\alpha] \\
\text{c.reduce(base, op)} & x_2 = \texttt{op}(x_1, e_\beta) & [t_\beta] \\
 & \textit{return } x_2 & [t_m]
\end{array}$$

Figure 3.2: **Model of collection operations.** The abstract thread executing each operation is bracketed to its right.

A *potential race* is a pair of accesses to the same field of the same object, such that one is a write access executed by t_α and the other is either a read or a write executed by t_β. In our example, there are several potential races on the `centerOfMass` field of the `NBodySimulation` object. $t''_\alpha : 25$ writes to the field `centerOfMass` while $t''_\beta : 24$ 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 `NBodySimulation` object are potentially racing. Accesses at lines 28, 30, and 31 in thread t''_β are also racing with $t''_\alpha : 25$ 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 4.2 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 [43].

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 $t'_\alpha : 12$ is identified as the same with the object accessed at $t''_\alpha : 21$, but different from the object initialized at $t'_\beta : 12$ (crossed arrow). Similarly, the object initialized at $t'_\beta : 12$ is the same with the object accessed at line $t''_\beta : 21$ 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 [59]. 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, out tool makes calls within parallel iterations context sensitive on the sharing nature of their arguments. Each call has a property shared in its context, with shared(argNo) meaning that the argNoth argument has not been instantiated in the current iteration. For the above example, shared(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 thread-safety model

IteRace uses a simple a priori thread-safety model of the classes to drastically reduce the number of warnings introduced by the intricate thread-safety mechanisms in libraries. To this purpose, we both adjust the context sensitivity and add one warning filtering phase.

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

- is threadSafe if any invocation of itself cannot be involved in races. All methods of thread-safe classes are at least threadSafe.
- is threadSafeOnClosure if it is threadSafe and any other invocation reachable from its invocation cannot be involved in races. This class of methods includes, but is not limited to, methods of immutable classes. As expected, all threadSafeOnClosure methods are also threadSafe. The converse is not true, as it is explained at the end of this subsection.
- instantiatesOnlySafeObjects if any object instantiated inside the method, but not necessarily in other methods called by it, is thread-safe.
- circulatesUnsafeObjects 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 *ThreadSafeOnClosure* sticky flag when the callee is threadSafeOnClosure.

Additionally, *Interesting* and *Uninteresting* sticky flags are used to indicate that the downstream call graph should always, respectively never, be expanded according to other rules (i.e. the ones introduced by *2-Threads* and *Bubble-up*).

The flags are 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:

- threadSafe(m_a)
- instantiatesOnlySafeObjects(m_o) – this is mostly useful for anonymous classes as they cannot be modeled with threadSafe
- the context of n_a is *ThreadSafeOnClosure*

It is possible to have methods that are threadSafe but not threadSafeOnClosure. Let us go back to the example in Fig.2.1. 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 threadSafe as a race cannot occur within it but it is not considered threadSafeOnClosure 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 if `String.valueOf(x)` were within the synchronized section, it wouldn't have helped, as another access holding a different lock or none at all could still race with it. The method also calls `print(String)` and `newLine()`. These methods are threadSafeOnClosure 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. 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.

For each race in library code, we have a pair of sets of application-level accesses leading to it. The sets are computed by traversing the call graph backwards, from the race to the first call graph node outside of library code.

Finally, IteRace groups warnings on each application-level receiver objects. 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. 2.1), 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.

The *Bubble-up* technique also adds a layer of object sensitivity between the application and library to improve precision. This layer is also sensitive to the presence of the *Interesting* flag described in Section 3.2.

3.4 Synchronized accesses

We determine locksets and filter races in a similar manner to Naik et al. [44]. Locks are represented by abstract objects. A lock protects an access if, for each each path through the program reaching the access, the last lock operation on the said lock is an acquisition. A pair of accesses is considered safe if the intersection of their locksets is not empty.

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. Our approach is similar to [44] 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.

Additionally, we filter safe accesses at two levels: once on an initial set of races, as in previous work [45], 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.

3.5 Discussion

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 threadSafe, threadSafeOnClosure, instantiatesOnlySafeObjects, or circulatesUnsafeObjects. 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, especially if she uses other libraries containing thread-safe classes.

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. 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.

Chapter 4

Evaluation

We evaluate our tool by answering the following questions:

1. **Is IteRace practical?** As the main culprit 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 warnings reported by IteRace 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 [44].

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 tandem with other techniques.

4.1 Methodology

We evaluate our approach by using IteRace to analyze the 7 open-source Java projects shown in Table 4.1. Then, we use JChord to analyze the same projects under the same conditions and compare the results. Finally, we measure the impact of each of our specialization techniques.

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 4.1 shows a short description of each application and indicates which part of

Table 4.1: **Evaluation suite.** 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.

	Description	SLOC (k)	#
Project	(parallel section)	(app+lib)	methods
mc	Monte Carlo simulation (the separate simulations) [16]	1.4 + 220	252
em	3D EM wave propagation simulation (force update) [17]	0.2 + 220	80
coref	NLP coreference finder (processing documents) [11]	41 + 225	927
weka	data mining software (generation of clusterers) [33]	301 + 253	1236
lucene	Lucene search benchmark (separate searches) [12]	48 +220	2363
junit	testing framework (jUnit's own test suite)	16 + 220	508
cilib	computational intelligence library (simulation engine)	53 + 454	1957

it is parallel, the application's size in lines of code, and the number of methods analyzed by our tool.

As Java 8 has not been released yet, analysis tools, including WALA, do not 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 [2] that exposes part of the new collection methods proposed for Java 8, but implemented with anonymous classes. 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 a profiler and identify the computationally intensive loop and the data structure it is iterating.

2. refactor the data structure into our 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.

IteRace 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.

At first, we only considered JDK as library code and, despite our techniques reducing the number of warnings by orders of magnitude, we still found ourselves needing to analyze a few thousands of warnings. Many of the warnings were still 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 a top-down approach based on our *Bubble-up* technique: We first aggressively mark 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, we gradually remove the library markings until the source for the race reveals itself. In our experiments, it took up to 10 analysis reruns in order to find the set of library markings that best describe the fault. For each application, it took us between a few minutes and a few hours to reach this optimal level. 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.

We also analyze all applications with selectively deactivating various techniques to reveal their effect upon the analysis as a whole. 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 3.4). 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.

JChord We also analyze all projects using JChord. We 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 warning are obviously false and are easy to filter out, so we considered it is 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 [43] as it is not currently available.

Additionally, we set JChord to ignore classes that IteRace models as threadSafeOnClosure and do not circulatesUnsafeObjects. This increases the tool's precision without hampering safety.

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.

4.2 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 4.2 shows an overview of the results. For context, the first three columns show JChord's performance analyzing the evaluation suite applications. JChord's runtime is reasonable but the reported number of warnings is overwhelming for five out of the seven case studies. For em3d and junit the number of warnings is low enough to be inspected but all of the warnings are false.

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, their results may vary in

Table 4.2: **Overall results.** "#" is the number of warnings. "real" is how many of the warnings are real races. 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.

	JChord			IteRace (our tool)			
		warnings			warnings		
project	t (s)	#	real	t (s)	#	real	faults
em3d	20	15	0	3.7	0	0	0
mc	22	44	1	5.4	1	1	0
junit	24	123	0	9.5	0	0	0
coref	85	19.5k	-	154.8	34	30	2
lucene	95	53.4k	-	171.9	119	2	2
weka	156	19.6k	-	432.2	1	1	1
cilib	271	21.4k	-	112.4	1735	2	1

terms of number of warnings. Still, JChord's results can give an idea about the effectiveness of a tool not implementing our techniques. JChord's results are similar to that of our tool with only the *Deep-Synchronized* technique activated.

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 3.5). Thus, it is highly unlikely that IteRace will miss any true races JChord finds.

The last four columns show IteRace's performance over the same applications. As expected, the runtime varies significantly with the size of the application, but it is acceptable even for the very large ones. For two applications, our tool doesn't report any races, correctly deeming them safe. For the other applications, after *Bubble-up*, the number of warnings is low and the reported accesses are close enough to the parallel loop body to be relatively easy to understand.

Furthermore, at first glance, the number of warnings might seem rather large. Still, the way IteRace reports them makes them easy to understand. 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 5 write (α) accesses and 10 β accesses would generate 50 race warnings, as counted in Table 4.2. Still, it is relatively easy for a programmer familiar with the application to inspect 5+10 accesses involving the same field of the same object.

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 34 warnings out of which 30 are true. 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 many warnings out of which two 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, we aim the analysis at various parts of its extensive algorithm library. For some algorithms, the analysis is very precise, reporting only two warnings, both true. We reported them to cilib developers and they confirmed and fixed the fault [1].

For other cilib algorithms, IteRace proved less precise, 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 makes the technique described at the end of Section 3.1 less effective when the receiver points to both shared and non-shared objects.

Effect of each specialization technique Tables 4.3 shows the runtime and Table 4.4 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 due to its limited impact (see the end of the section) and space constraints. 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

Table 4.3: **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.**
				3.5	4.0	5.7	22.3	16.9	45.4	22.3	**11.8**
			●	4.0	5.4	7.6	88.3	87.6	191.4	62.1	**28.5**
		●		3.6	5.1	6.8	470.7	469.3	302.8	45.5	**45.2**
		●	●	3.9	6.0	8.1	723.3	582.0	429.5	75.8	**59.6**
	●			3.7	4.3	6.7	35.8	29.3	91.9	26.4	**16.0**
	●		●	3.6	4.8	8.2	62.5	62.8	147.2	47.2	**23.4**
	●	●		3.7	4.4	6.8	35.5	34.5	91.7	27.6	**16.7**
	●	●	●	3.9	5.0	8.3	60.6	63.0	147.5	47.0	**23.8**
●				3.7	4.2	6.2	62.3	36.9	75.8	38.5	**18.2**
●			●	3.7	5.5	7.9	271.8	175.9	492.3	145.6	**47.9**
●		●		3.7	4.3	6.3	86.0	68.3	172.9	51.9	**24.6**
●		●	●	3.2	5.4	8.1	247.9	183.6	541.2	148.9	**47.3**
●	●			3.8	4.3	7.1	76.6	70.0	221.9	54.1	**25.9**
●	●		●	3.7	5.5	9.5	145.6	159.4	427.8	113.3	**41.8**
●	●	●		3.4	4.6	7.2	75.8	86.6	240.3	60.0	**27.2**
●	●	●	●	3.7	5.4	9.5	154.8	171.9	432.2	112.4	**42.5**

Table 4.4: **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	2541	2389	81K	151K	110K	71K
			●	1	2541	2351	81K	151K	103K	42K
		●		1	748	222	586K	246K	20K	11K
		●	●	1	748	203	586K	244K	20K	11K
	●			1	179	49	22K	37K	6675	9447
	●		●	1	179	24	22K	37K	6602	9442
	●	●		1	155	36	476	8312	1344	2771
	●	●	●	1	155	30	476	6425	1344	2762
●				0	53	87	22K	32K	38K	38K
●			●	0	53	70	21K	30K	32K	18K
●		●		0	3	3	36K	13K	10K	6293
●		●	●	0	3	0	36K	12K	10K	6251
●	●			0	1	17	427	14K	472	1795
●	●		●	0	1	0	427	12K	463	1791
●	●	●		0	1	3	34	2006	1	1741
●	●	●	●	**0**	**1**	**0**	**34**	**119**	**1**	**1735**

Table 4.5: **Effect of *2-Threads* on the number of warnings.**
(improvement ratio, see third paragraph of Sec. 4.2)

F	B	S	em3d	mc	junit	coref	lucene	weka	cilib
			∞	47.94	27.46	3.70	4.71	2.85	1.86
		•	∞	47.94	33.59	3.70	5.02	3.18	2.31
	•		∞	249.33	74.00	15.97	17.73	1.95	1.77
	•	•	∞	249.33	∞	15.97	20.35	1.95	1.77
•			∞	179.00	2.88	53.13	2.66	14.14	5.26
•		•	∞	179.00	∞	53.12	2.94	14.26	5.27
•	•		∞	155.00	12.00	14.00	4.14	1344.00	1.59
•	•	•	∞	155.00	∞	14.00	53.99	1344.00	1.59

techniques are activated (last row of Table 4.4). IteRace finishes the analysis in under two minutes for all applications except WEKA.

Tables 4.5, 4.6, 4.7, and 4.8 highlight the effect of activating/deactivating each technique. These tables are derived from Table 4.4. 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* deactivated) and the "junit" column in Table 4.5 is obtained from Table 4.4, 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 4.5 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 4.6 shows that *Filtering* has a powerful effect for all larger applications. The filtered out warnings mostly involve accesses to library classes, e.g., synchronized I/O, Java security, regex, and concurrent or synchronized collections.

Table 4.7 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

Table 4.6: **Effect of *Filtering* on the number of warnings.** (improvement ratio, see third paragraph of Sec. 4.2)

T	B	S	em3d	mc	junit	coref	lucene	weka	cilib
			1.00	14.20	48.76	3.59	4.05	16.63	7.59
		●	1.00	14.20	97.96	3.59	4.08	15.62	4.45
	●		1.00	4.83	6.17	1233.12	29.70	15.56	4.01
	●	●	1.00	4.83	6.77	1233.12	38.13	15.56	4.01
●			NaN	53.00	5.12	51.56	2.28	82.42	21.47
●		●	NaN	53.00	∞	51.47	2.38	70.10	10.17
●	●		NaN	3.00	1.00	1081.03	6.94	10711.00	3.61
●	●	●	NaN	3.00	NaN	1081.03	101.16	10711.00	3.60

Table 4.7: **Effect of *Bubble-up* on the number of warnings.** (improvement ratio, see third paragraph of Sec. 4.2)

T	F	S	em3d	mc	junit	coref	lucene	weka	cilib
			1.00	3.40	10.76	0.14	0.61	5.31	6.45
		●	1.00	3.40	11.58	0.14	0.62	4.93	3.79
	●		1.00	1.15	1.36	47.66	4.50	4.97	3.41
	●	●	1.00	1.15	0.80	47.65	5.77	4.91	3.42
●			NaN	17.67	29.00	0.60	2.31	3.63	6.12
●		●	NaN	17.67	∞	0.60	2.50	3.03	2.91
●	●		NaN	1.00	5.67	12.56	7.02	472.00	1.03
●	●	●	NaN	1.00	NaN	12.56	106.08	463.00	1.03

Table 4.8: **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	1.00	1.08	1.71
		●	1.00	1.00	1.09	1.00	1.01	1.00	1.00
	●		1.00	1.00	2.04	1.00	1.01	1.01	1.00
	●	●	1.00	1.00	1.20	1.00	1.29	1.00	1.00
●			NaN	1.00	1.24	1.00	1.07	1.20	2.12
●		●	NaN	1.00	∞	1.00	1.16	1.00	1.01
●	●		NaN	1.00	∞	1.00	1.12	1.02	1.00
●	●	●	NaN	1.00	∞	1.00	16.86	1.00	1.00

technique maps deep warnings into a 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. This effect is revealed by the subunitary 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 4.8 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*.

Chapter 5

Related work

5.1 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 [9, 21, 22, 25, 41, 55, 58, 61]. This approach can miss many races so lockset-based race detectors have been developed as an alternative that catches more races at the expense of false positives [19, 46, 57, 62]. There are also hybrid approaches that combine both techniques [18, 30, 49, 65].

Similarly, static race detectors vary between higher precision, lower scalability [35, 43] and lower precision, better scalability [38, 44, 50, 51, 64]. Also, annotations can be used to improve the performance of the analysis [8].

5.2 Static analyses for C and other languages

Several race analyses have been proposed for C or variants [26,31,52]. Henzinger et al. [35] present a model checking approach that is both path and flow sensitive, and models thread contexts. Pratikakis et al. present LOCKSMITH [50, 51], a type-based analysis that computes context-senstitive *correlations* between lock and memory accesses. RELAY [64] 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.

5.3 Static analyses for Java

Flanagan et al. [27] proposed using type checking systems to find races. Boyapati et al. [14, 15] 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 [10, 28, 29, 56].

Praun et al. [63] propose an Object Use Graph model that statically approximates the happens-before relation between accesses to a specific object.

Choi et al. [20] 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. [44] builds an object-sensitive analysis that uses thread-escape to lower the false positive rate. In a subsequent article [43], they present a conditional must not alias analysis for solving aliasing relationships between locks.

Chapter 6

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 will be 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 [23, 24, 32], that require fast and precise analyses.

References

[1] CIlib bug. https://github.com/cilib/cilib/issues/111.

[2] Concurrency JSR-166 Interest Site - ParallelArray. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[3] JDK8. http://jdk8.java.net.

[4] Microsoft TPL. http://msdn.microsoft.com/en-us/library/dd460717.aspx.

[5] State of the Lambda: Libraries Edition. http://cr.openjdk.java.net/ briangoetz/lambda/sotc3.html.

[6] Threading Building Blocks. http://threadingbuildingblocks.org/.

[7] WALA documentation. http://wala.sourceforge.net/.

[8] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *TOPLAS*, 28:207–255, March 2006.

[9] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19:234–243, April 1991.

[10] Rahul Agarwal and Scott Stoller. Type inference for parameterized race-free Java. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 77–108. Springer Berlin / Heidelberg, 2004.

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

[12] 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. 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, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[13] Eric Bodden and Klaus Havelund. Racer: effective race detection using AspectJ. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 155–166, New York, NY, USA, 2008. ACM.

[14] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[15] Chandrasekhar Boyapati and Martin Rinard. 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, pages 56–69, New York, NY, USA, 2001. ACM.

[16] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. In *Java Grande*, pages 81–88. ACM Press, 1999.

[17] B. Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 280 –291, 2001.

[18] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for Java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 221–230, New York, NY, USA, 2008. ACM.

[19] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 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, pages 258–269, New York, NY, USA, 2002. ACM.

[20] Jong-Deok Choi, Alexey Loginov, and Vivek Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.

[21] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *TOPLAS*, 13:491–530, October 1991.

[22] Mark Christiaens and Koen De Bosschere. 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, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[23] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. 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, pages 793–794, New York, NY, USA, 2009. ACM.

[25] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. *SIGPLAN Not.*, 25:1–10, February 1990.

[26] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252, October 2003.

[27] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[28] Cormac Flanagan and Stephen N. Freund. 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, pages 90–96, New York, NY, USA, 2001. ACM.

[29] Cormac Flanagan and Stephen N. Freund. Type inference against races. *Sci. Comput. Program.*, 64:140–165, January 2007.

[30] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[31] Dan Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 13–25, New York, NY, USA, 2003. ACM.

[32] Alex Gyori, Danny Dig, Lyle Franklin, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '13, 2013.

[33] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[34] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 353–364, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 1–13, New York, NY, USA, 2004. ACM.

[36] Michael Hind. 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, pages 54–61, New York, NY, USA, 2001. ACM.

[37] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. *SIGPLAN Not.*, 42:339–350, January 2007.

[38] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, ESEC/FSE '09, pages 13–22, New York, NY, USA, 2009. ACM.

[39] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. 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, pages 411–427, New York, NY, USA, 2010. ACM.

[40] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[41] John Mellor-Crummey. 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, pages 24–33, New York, NY, USA, 1991. ACM.

[42] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005.

[43] Mayur Naik and Alex Aiken. 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, pages 327–338, New York, NY, USA, 2007. ACM.

[44] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

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

[46] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *VM*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[48] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ESEC/FSE '12, pages 54–65, New York, NY, USA, 2012. ACM.

[49] Eli Pozniansky and Assaf Schuster. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[50] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

[51] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.*, 33:3:1–3:55, January 2011.

[52] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

[53] Cosmin Radoi and Danny Dig. Practical static race detection for java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 178–190, New York, NY, USA, 2013. ACM.

[54] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[55] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17:133–152, May 1999.

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

[57] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[58] D. Schonberg. On-the-fly detection of access anomalies. *SIGPLAN Not.*, 24:285–297, June 1989.

[59] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 1981.

[60] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410, New York, NY, USA, 2011. ACM.

[61] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[62] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 70–82, New York, NY, USA, 2001. ACM.

[63] Christoph von Praun and Thomas R. Gross. 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, pages 115–128, 2003.

[64] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[65] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39:221–234, October 2005.