Practical static race detection for Java parallel loops

Cosmin Radoi
University of Illinois
cos@illinois.edu

Danny Dig
University of Illinois
dig@illinois.edu

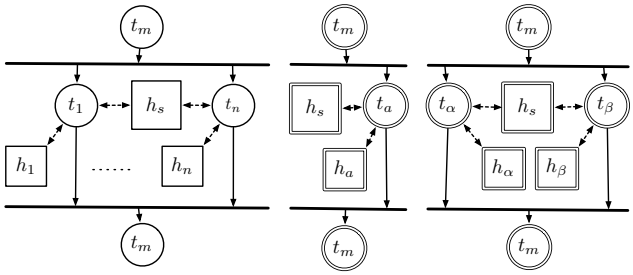
***Abstract*—Despite significant progress in recent years, the important problem of static race detection remains open. Previous techniques took a *general* approach and looked for races by analyzing the effects induced by low-level concurrency constructs (e.g., java.lang.Thread). But constructs and 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 Java8. Our evaluation shows that ITERACE is efficient, running in under 100 seconds even for programs of hundreds of thousands of lines of code. Also, it is precise, reporting no false positives in 5 out of the 7 case studies.

I. 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 [1]. The major programming languages have parallel constructs or libraries that support loop parallelism very well, e.g., `Parallel.For` in .NET TPL [2], `.parallel()` in the upcoming Java8 [3], `parallel_for` in C++ TBB [4]. 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 manually due to non-deterministic thread scheduling. This has led to a large body of research on race detection. Static race detection techniques [5]–[16] use an underlying static modeling of the program's real execution. This static approach allows a single analysis pass to find all the races that could occur in many possible program executions. Static race detectors rarely miss races but are faced with the opposite problem: despite continuous improvements, they still report very many false warnings. For example, we applied JChord [7], the current state-of-the-art, on compute-intensive loops from seven Java real programs. JChord reports on average 5740 racing accesses per analyzed loop. This can 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 [14], [15], [17]–[30], which typically have much fewer false warnings, but have high overhead and do not expose races on program paths that are not executed.



(a) Runtime (b) General (c) ITERACE

Fig. 1: Modeling of a parallel loop. Circles are threads, squares are parts of the heap. Double line denotes abstraction.

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*: being efficient and reporting a low 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.

We present three *specialization* techniques that each can lower the number of false warnings: (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 classes. We implemented these ideas in a tool, ITERACE, and empirically validated how well they work individually, and in tandem.

1) 2-Threads: A parallel loop is an SPMD-style (Single Program, Multiple Data) computation. Its iterations are identical tasks processing different input. 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 parallel loop 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, e.g., objects created by thread t_i.

A general race detector models the identical forked threads by only one abstract thread [7], [16] (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, we propose a specialized technique that 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 shared abstract objects 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 escapes, our analysis does not implicitly consider it unsafe. ITERACE only reports a race when the object reaches the other abstract thread and there is a concurrent access.

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 care 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: such a race is rarely due to a buggy library; it is more 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. In a sense, the application-level race warnings can be seen as atomicity violation warnings on using library code.

3) Filtering: To improve performance, some library classes employ advanced synchronization techniques (e.g., memory fences, compare-and-swap, spin-locks, 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 the advice in [31] on the importance of client-specific pointer analysis, we use this model to specialize the context sensitivity to increase precision and lower runtime.

This paper makes the following contributions:

- **Race detection approach.** We propose three techniques for making the static race detection of applications practical. Our approach (i) *specializes* to lambda-style parallel loop operations, (ii) traces, summarizes, and reports the race warnings in application code, and (iii) is aware of and uses thread-safety properties of classes
- **Tool.** We implemented these techniques in a tool, ITERACE, that analyzes Java programs and released it as open-source: http://github.com/cos/IteRace
- **Evaluation.** We evaluated our approach by using ITERACE to analyze 7 open-source projects. We also analyzed the same projects with a state-of-the-art, but general, static race detection tool, JChord [7]. The results show that our specialized approach is fast, running in under 100 seconds even for programs of hundreds of thousands of lines of code. Also, it is precise, reporting no false positives in 5 out of the 7 case studies. We designed and carried out a set of experiments to measure the effect of each specialization technique alone and in tandem with other techniques.

II. 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 to be introduced in Java8 [32]. In Java8, 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 operation defined 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. Still, as shown in the evaluation, our tool analyzed correctly, without false positives, the Barnes-Hut implementation of `updateForce`. 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

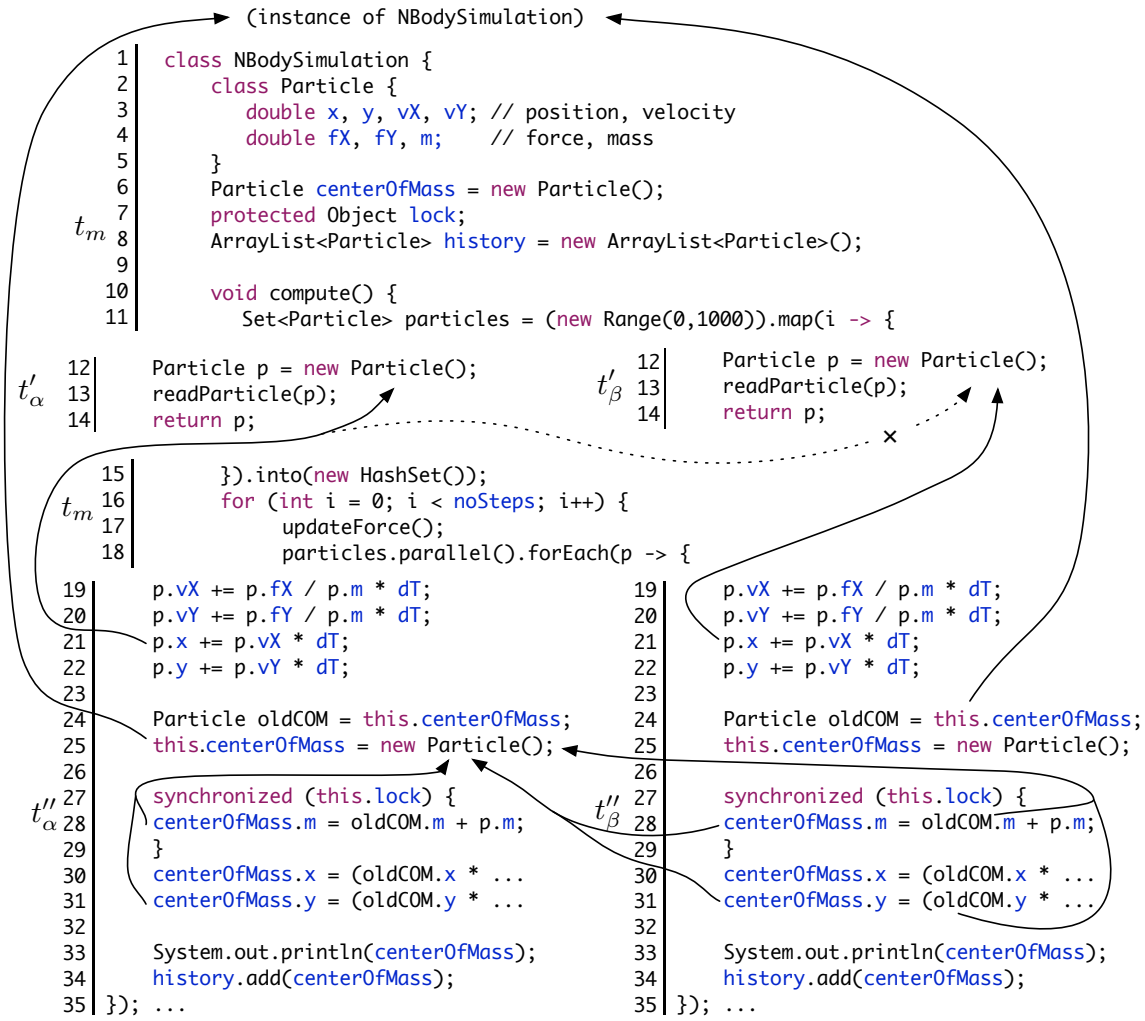


Fig. 2: Visual representation of how our analysis sees a simple N-body simulation implementation. The blocks of code are labeled with the abstract thread that executes them, e.g., t_α. The arrows show points-to relations from variables to allocation sites, e.g., variable `p` at line 21 in thread t''_α 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.

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 `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, *App-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.

III. RACE DETECTION

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

Figure 3 presents a high level overview of ITERACE. WALA [33] 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

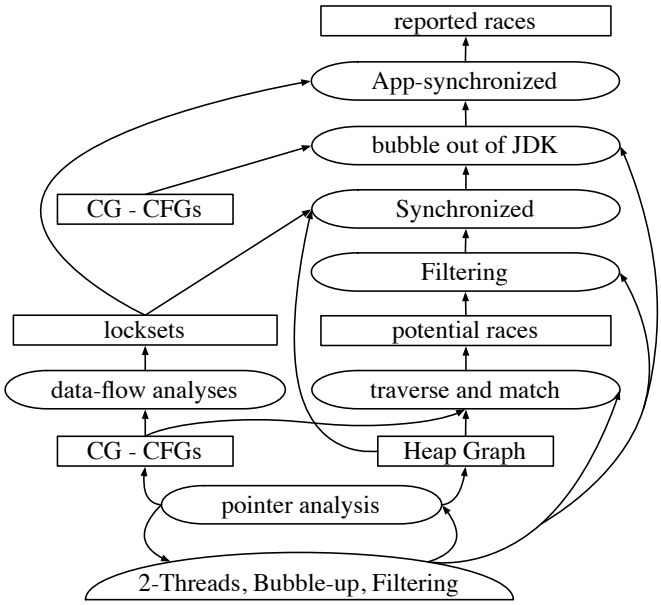


Fig. 3: Analysis overview. Ovals represent different sub-analyses. Rectangles represent intermediate and final data structures. The bottom half-oval represents the specialized context sensitivity mechanism.

in sections III-A, III-D, and III-B. 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. 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. III-A).

Also, for each statement in the program, ITERACE computes the lock set that protects it. This is achieved by two data-flow analyses, one intra-procedural and one inter-procedural.

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

Accesses protected by the same lock are race-free. Thus, the *Synchronized* phase (Sec. III-C) filters out the potential races containing 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. III-D).

Finally, *App-synchronized*, a stage similar to *Synchronized*, further prunes the bubbled-up race warnings.

A. 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, the concrete threads executing each loop 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

c.forEach(op)	$\texttt{op}(e_\alpha)$	$[t_\alpha]$
	$\texttt{op}(e_\beta)$	$[t_\beta]$
c.map(op)	$e_\alpha = \texttt{op}(e_\alpha)$	$[t_\alpha]$
	$e_\beta = \texttt{op}(e_\beta)$	$[t_\beta]$
	$return\ c$	$[t_m]$
reduce(base, op)	$x_1 = \texttt{op}(e_\alpha, \text{base})$	$[t_\alpha]$
	$x_2 = \texttt{op}(x_1, e_\beta)$	$[t_\beta]$
	$return\ x_2$	$[t_m]$

Fig. 4: Abstract model for collection operations. The abstract thread executing each operation is bracketed to its right.

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 [34] 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. 4 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 modeling does not respect the laziness characteristic of some of the new collection operations [3], this does not affect the results of the race detection analysis.

The above modeling is used for both the parallel and the sequential loop operations over the collection of interest. This allows the tool to understand the relationships between the elements of the collection as they are processed by different loops. In Figure 2, both the collection initialization at lines 11-15 and the processing at lines 18-35 are modeled. Therefore, ITERACE sees that the element `p` in t''_α is the same with the element `p` in t'_α but different from the element `p` from t'_β.

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 IV-B 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 variable is escaping through `particles`. Then, other more expensive analyses could be further employed to refine the results, for example [10].

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.

B. Filtering using thread-safety model

ITERACE uses a simple a priori thread-safety model of the classes to drastically reduce the number of warnings. We both adjust the context sensitivity and add one filtering phase.

Filtering uses the following a priori information about methods. Thus, 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 an unsafe object as a parameter.

Using the above information, the context of a callee is generated from the context of the caller by the following rules, executed in order:

- add a *ThreadSafeOnClosure* sticky flag when the called method is threadSafeOnClosure and the calling context is not *Uninteresting*
- add an *Interesting* sticky flag when the called method circulatesUnsafeObjects and the calling context is not *Uninteresting*
- replace all context information with a singleton *Uninteresting* context when the calling context, also modified by the above rules, is *ThreadSafeOnClosure* and is not *Interesting* . This context is a "black hole" for code that will not be interesting in any way for race detection.

The *ThreadSafeOnClosure* and *Interesting* 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 one 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. 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 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.

C. Synchronized accesses

We determine locksets and filter races in a similar manner to Naik et al. [7]. The differentiating aspect is that we apply the algorithm twice, once on an initial set of races, as in Naik's work, and once after the *Bubble-up*. Our evaluation shows that applying the algorithm after *Bubble-up* is much more effective.

D. 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 data 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 on 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 at the application level.

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 III-B.

E. Discussion

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.

Also, ITERACE is designed to analyze the loop-parallel parts of the program and cannot reason about concurrency that appears by spawning other threads besides the ones spawned by the parallel loops. In such cases, ITERACE warns the programmer about the potentially unsafe thread spawn.

May-alias lock equality: Using may-alias information to approximate must-alias lock relations is unsafe. Still, our analysis could easily be adapted to use a must-alias analysis with just a drop-in replacement. Also, our evaluation shows that the *Synchronized* and *App-synchronized* phases have much less warning-reduction effect than the others so the programmer could choose to deactivate these phases to get safer results.

IV. EVALUATION

We evaluate our tool by answering the following questions:

1) How does ITERACE compare with the state of the art? We compare ITERACE with a state of the art, but general, data race detection tool for Java, JCHORD [7].
2) How much does each specialization technique contribute to ITERACE's efficiency and precision? We not only measure the performance of the tool as a whole, but also the effect of each specialization technique, as applied individually and in tandem with other techniques. We gauge efficiency and precision by running time and number of warnings, respectively.

A. Methodology

We evaluate our approach by using ITERACE to analyze 7 open-source Java projects shown in Table I. 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 upon the efficiency and precision of the tool as a whole.

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 `Thread`-based implementations of loop parallelism to improve the performance of their application, i.e., Barnes-Hut, Lucene, and jUnit. Thus, we looked further to applications that have sequential implementations but where the underlying algorithm is inherently parallel and included five more applications, i.e., MonteCarlo, EM3D, Coref, and Weka.

The evaluation suite is heterogenous: it has applications from different domains (benchmarks, NLP, data mining, testing) and of various sizes, from hundreds of lines of code to tens of thousands (see Table I).

As Java8 will only be released in 2013, 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 [37] that exposes part of the new collection methods proposed for Java8, but implemented with anonymous

Project	Description	SLOC (app+lib)	# Methods	Parallelized
BH	Barnes-Hut simulation	899+220k	865	force update step [35]
MC	Monte Carlo simulation	1441+220k	252	the separate deterministic computations [36]
EM3D	electromagnetic wave propagation simulation	181+220k	80	force update for nodes
Coref	NLP coreference finder	41k+225k	927	processing documents
Weka	data mining software	301k+253k	1236	generation of clusterers
Lucene	Lucene search benchmark	48k+220k	2363	separate searches
jUnit	testing framework	15.6k+220k	508	jUnit's own test suite
Average		58k+225k	890	

TABLE I: Evaluation suite. Column 4 shows the number of methods analyzed by ITERACE. The size of library code varies as some programs use extra libraries besides JDK.

	EM3D	BH	MC	jUnit	Coref	Lucene	WEKA	**Mean**		EM3D	BH	MC	jUnit	Coref	Lucene	WEKA	**Mean**
`tfbs`	7.0	15.5	9.9	11.4	28.4	35.5	34.3	**17.1**		4	81	3783	3541	387428	91182	118922	**4061**
`tfbS`	7.1	15.1	15.4	14.6	148.1	74.6	89.5	**30.4**		4	81	3783	3508	387246	91151	118801	**4055**
`tfBs`	6.9	16.0	16.5	15.8	87.4	48.7	508.4	**34.9**		4	81	42528	17915	13.4M	6.0M	21.7M	**46033**
`tfBS`	7.0	16.1	27.4	19.6	-	-	-	**74.9**		4	81	42375	17302	-	-	-	**45781**
`tFbs`	6.8	20.5	9.3	10.6	33.5	34.5	38.0	**18.0**		4	81	202	59	11686	33464	8974	**540**
`tFbS`	7.0	20.1	10.6	11.7	39.1	52.9	52.5	**21.2**		4	81	202	23	11682	33445	8853	**471**
`tFBs`	6.8	19.4	9.7	10.6	37.0	38.7	84.3	**20.7**		4	81	186	96	10176	25931	16025	**588**
`tFBS`	7.0	19.1	10.6	11.7	43.3	54.6	124.7	**24.3**		4	81	178	63	5551	18364	9424	**445**
`Tfbs`	6.9	14.7	10.4	11.3	28.5	34.9	34.8	**17.0**		0	0	3246	3119	258708	45154	104473	**1174**
`TfbS`	6.6	15.9	14.6	14.4	84.9	58.9	77.3	**26.3**		0	0	3246	3095	258673	45136	104356	**1173**
`TfBs`	7.0	15.8	17.4	17.1	81.8	47.3	463.7	**34.6**		0	0	42081	17767	13.4M	6.0M	21.5M	**16453**
`TfBS`	6.8	15.8	26.8	19.6	-	-	-	**74.2**		0	0	41955	17203	-	-	-	**16407**
`TFbs`	6.5	19.0	9.5	10.6	33.1	35.0	35.6	**17.6**		0	0	3	24	2372	12030	1022	**47**
`TFbS`	7.0	19.3	10.5	11.7	39.0	49.5	64.7	**21.4**		0	0	3	0	2370	12021	1009	**27**
`TFBs`	6.9	20.5	9.5	10.8	35.4	38.6	84.8	**20.8**		0	0	1	4	1092	9081	7328	**35**
`TFBS`	6.9	20.7	10.6	12.6	42.2	53.1	124.2	**24.5**		0	0	1	0	1029	5049	3417	**21**

TABLE II: Evaluation results - left: runtime (seconds); right: number of race warnings, i.e., racing pairs of accesses.
t - *2-Threads*, f - *Filtering*, b - *Bubble-up*, s - *App-synchronized*; upper case denotes an activated feature; dash - out of memory

classes. Once tools like WALA handle lambda expressions, adapting the implementation will be trivial.

For already-parallelized 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.
4) mark all variables local to the method that were read within a refactored loop with the "final" keyword, and transform all local variables that were written within a refactored loop to a one element array (to overcome Java's limitation requiring local variables in a closure to be final)

2) 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 different abstract threads. As JChord 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 more fair towards JChord to disregard them.
- it ignores races in constructor code. This significantly reduces the number of false positives reported by JCHORD but adds another source of unsoundness. While rare, a race can involve constructor code, e.g., a constructor reads a field of an object while another thread writes that field. ITERACE does not ignore races in constructors.
- it does not use conditional must not alias analysis [10] as it is not currently available.

3) ITERACE*:* We analyze all applications using ITERACE both with all techniques activated and 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 races that are correctly synchronized, both in the classical way and at application level. 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 Intel Core 2 Quad at 2.4 GHz (Q6600) with 4 GB of RAM. The JVM is allocated 2 GB of RAM. We implemented the race-detection techniques in Scala and we use the static analysis framework WALA, which is implemented in Java.

B. Results

Table II shows the runtime and the number of races reported by our analysis under 16 of the 32 possible configurations. We are not showing results for filtering races 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 labeled by an acronym where an upper/lower case denotes a technique is activated/deactivated. The mean in this table and all following tables is the geometric mean.

The results confirm that our analysis is effective. By far, the best results are obtained when all techniques are activated (last row of Table II). ITERACE finishes the analysis in under one minute for all applications except WEKA. For three out of the seven projects, ITERACE correctly reported no race warnings.

1) Comparison with JChord: Table III shows a comparison between JChord and the best configuration of ITERACE when analyzing the loop-parallel programs in the evaluation suite.

ITERACE is faster in analyzing all programs, except WEKA and jUnit. Still, the speed difference can be subjective as the two analyses are based on different underlying pointer analysis implementations, with the pointer analysis accounting for a large proportion of the computation in both cases.

The only application JCHORD performs as well as ITERACE is MC. In all other applications, ITERACE reports a significantly lower number of warnings. In order to better gauge the efficacy of the tools, we also estimate the rate of false positives. For this, we have asked two graduate students, who are not involved with the project and have good concurrent programming experience, to inspect a sample of race reports from both tools. For JCHORD they sampled 20 racing pairs for each project (where possible – MC and EM3D have fewer warnings). For ITERACE, they sampled 10 race sets, consisting of several dozens racing pairs, for each project. Column "% real" summarizes the results of their inspection.

BH, EM3D, and jUnit are race free and ITERACE correctly reports no warnings for any of them, while JCHORD reports false warnings for each of them. MC contains a benign race reported by both tools.

For Lucene, both tools report a very high number of warnings. While most of the warnings are false, both tools identify a true race: there is an unsynchronized concurrent static access to `java.text.DateFormat` in the `getRangeQuery` method of the `QueryParser` class. The four percent difference between JCHORD's and ITERACE's true positive rate is an artifact of the sampling method.

	JCHORD			ITERACE			
		warnings			warnings		
project	t (s)	#	% real	t (s)	#	sets	% real
BH	33	108	0	20.7	0	0	-
MC	25	1	100	10.6	1	1	100
EM3D	19	15	0	6.9	0	0	-
Coref	56	4112	0	42.2	1029	91	100
Weka	85	5890	20	124.2	3417	47	95
Lucene	139	37138	5	53.1	5049	214	1
jUnit	10	167	0	12.6	0	0	-

TABLE III: Comparison with JChord.

t*-T*	EM3D	BH	MC	jUnit	Coref	Lucene	WEKA	**Mean**
`fbs`	4	81	537	422	128720	46028	14449	**2886**
`fbS`	4	81	537	413	128573	46015	14445	**2881**
`fBs`	4	81	447	148	29095	32574	184437	**29580**
`fBS`	4	81	420	99	-	-	-	**29373**
`Fbs`	4	81	199	35	9314	21434	7952	**493**
`FbS`	4	81	199	23	9312	21424	7844	**444**
`FBs`	4	81	185	92	9084	16850	8697	**552**
`FBS`	4	81	177	63	4522	13315	6007	**424**
Mean	**3**	**80**	**1416**	**521**	**91888**	**73066**	**71392**	

TABLE IV: Effect of *2-Threads* - number of races.

For WEKA, all threads share the same object of class `Instances`, generating many race conditions which are correctly reported by both tools. JCHORD also reports many false warnings while ITERACE reports very few.

Coref is one of the applications that we parallelized ourselves and the developers of the project told us that there is no interaction between the iterations of the parallelized loop that they know of. ITERACE actually reported many warnings which the developers then confirmed as true races.

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 five write (α) accesses and 15 β accesses contributes 5×15 race warnings to Table II. Still, it is relatively easy for a programmer familiar with the application to inspect $5+15$ accesses involving the same field of the same object. Column "sets" in table III shows the number of race sets reported for each of the applications. While Lucene was problematic, we found inspecting the reported race sets for Coref and WEKA easy and fast.

2) Effect of each specialization technique: Tables IV, V, VI highlight the effect of activating/deactivating each technique. The value in each cell is the difference between the number of races on a certain configuration with the technique deactivated and the number of races with the technique activated. Thus, a higher number means the technique filters out more false warnings.

Table IV shows that modeling each loop with two distinct threads significantly improves the results independent of other features. As expected, the filtered out accesses are on objects

f*-F*	EM3D	BH	MC	jUnit	Coref	Lucene	WEKA	Mean
tbs	0	0	3581	3482	375742	57718	109948	**3520**
tbS	0	0	3581	3485	375564	57706	109948	**3583**
tBs	0	0	42342	17819	13.4M	6.0M	21.7M	**45444**
tBS	0	0	42197	17239	-	-	-	**45335**
Tbs	0	0	3243	3095	256336	33124	103451	**1127**
TbS	0	0	3243	3095	256303	33115	103347	**1146**
TBs	0	0	42080	17763	13.4M	6.0M	21.5M	**16417**
TBS	0	0	41954	17203	-	-	-	**16386**
Mean	**0**	**0**	**12147**	**7603**	**2.0M**	**0.6M**	**1.5M**	

TABLE V: Effect of *Filtering* - number of races.

b*-B*	EM3D	BH	MC	jUnit	Coref	Lucene	WEKA	Mean
tfs	0	0	-38745	-14374	–13.1M	-6.0M	-21.6M	**-41971**
tfS	0	0	-38592	-13794	-	-	-	**-41726**
tFs	0	0	16	-37	1510	7533	-7051	**-47**
tFS	0	0	24	-40	6131	15081	-571	**26**
Tfs	0	0	-38835	-14648	-13.2M	-6.0M	-21.5M	**-15278**
TfS	0	0	-38709	-14108	-	-	-	**-15234**
TFs	0	0	2	20	1280	2949	-6306	**11**
TFS	0	0	2	0	1341	6972	-2408	**5**
Mean	**0**	**0**	**-461**	**-232**	**-0.15M**	**-0.24M**	**-0.39M**	

TABLE VI: Effect of *Bubble-up* - number or races.

that are thread-local by being either created and not escaped from the current iteration or unique to each element of the collection. While its effect is smaller than that of the *Filtering* technique discussed below, it is high enough to make the difference between a usable and an unusable tool.

Filtering, highlighted in Table V, 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.

The effect of *Bubble-up* on the number of warnings is highlighted in Table VI. Its main value is not in reducing the number of warnings but in making them more programmer friendly. As the technique maps a set of deep races into a set of application-level ones and as it is common for one library class to be used repeatedly throughout the application, the number of warnings is inflated. This effect can be seen in rows 1,2,5, and 6. Still, when combined with *Filtering* (rows 3,4,7, and 8) the negative effect is reversed and we can actually see improvement in most cases. This is because most of the warnings come from correctly-synchronized library classes.

In a similar manner to Tables IV-VI, each cell in Table VII shows the effect of analyzing synchronization at (i) deep, (ii) application, and (iii) both levels. We observe it is more effective to analyze synchronization at application level than at deep level. Analyzing at both levels brings almost no improvement while consuming time.

	EM3D	BH	MC	jUnit	Coref	Lucene	WEKA
tfb	0/0/	0/0/	0/0/	33/33/	182/182/	31/31/	121/121/
	0	0	0	33	182	31	121
tfB	0/0/0	0/0/0	0/153/153	9/613/613	0/-/-	17/-/-	-/-/-
tFb	0/0/	0/0/	0/0/	36/36/	4/4/	19/19/	121/121/
	0	0	0	36	4	19	121
tFB	0/0/	0/0/	0/8/	6/33/	2/4625/	7/7567/	10/6601/
	0	0	8	33	4626	7567	6602
Tfb	0/0/	0/0/	0/0/	24/24/	35/35/	18/18/	117/117/
	0	0	0	24	35	18	117
TfB	0/0/0	0/0/0	0/126/126	4/564/564	0/-/-	4/-/-	-/-/-
TFb	0/0/0	0/0/0	0/0/0	24/24/24	2/2/2	9/9/9	13/13/13
TFB	0/0/	0/0/	0/0/	4/4/	0/63/	4/4032/	0/3911/
	0	0	0	4	63	4032	3911

TABLE VII: Effect of analyzing synchronization at deep/application/both levels - number of races

V. Related work

A. Dynamic analyses

Dynamic race detectors have been the favored approach in the last decade. Their main advantage over static approaches is the typically lower number of false warnings. This advantage is counterbalanced by dynamic analyses' failure to catch races that are not exposed by the analyzed execution and the high computational cost of the more precise tools. The more precise dynamic race detectors compute happens-before relations over the events of an observed execution trace and, based on these relations, infer race conditions [17]–[21], [23], [24]. This approach is highly expensive so lockset-based race detectors have been developed as an alternative that trades precision for better performance [22], [25], [27]. There are also hybrid approaches that combine both techniques [29], [38]–[40].

Similarly, static race detectors vary between higher precision, lower scalability [5], [10] and lower precision, better scalability [7], [8], [13], [16], [41]. Also, annotations can be used to improve the performance of the analysis [6].

B. Static analyses for C and other languages

Several race analyses have been proposed for C or variants [5], [42]–[44]. More recently, Pratikakis et al. present LOCKSMITH [16], [41], a type-based analysis that computes context-senstitive *correlations* between lock and memory accesses. RELAY [8] 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.

C. Static analyses for Java

Flanagan et al. [45] proposed using type checking systems to find races. Boyapati et al. [46], [47] 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 [48]–[51].

Praun et al. [52] propose an Object Use Graph model that statically approximates the happens-before relation between accesses to a specific object.

Choi et al. [53] 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. [7] builds an object-sensitive analysis that uses thread-escape to lower the false positive rate. In a subsequent article [10], they present a conditional must not alias analysis for solving aliasing relationships between locks.

VI. 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 Java8. 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 and does not hinder the programmer with many warnings. Thus, ITERACE can also be used in scenarios with high interactivity, e.g., refactoring, that require fast and precise analyses.

ACKNOWLEDGMENTS

We thank Codruta Girlea, Stas Negara, Sandro Badame, Francesco Sorrentino, Rajesh Karmani, Nicholas Chen, Semih Okur, Samira Tasharofi, Milos Gligoric, Darko Marinov, and Vikram Adve for their feedback on earlier drafts of this work. We also thank Mihai Codoban and Caius Brindescu for their help in making an unbiased evaluation. This research was funded through two 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.

REFERENCES

[1] S. Okur and D. Dig, "How do developers use parallel libraries?" in *ESEC/FSE*, 2012.
[2] [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460717.aspx
[3] [Online]. Available: http://cr.openjdk.java.net/~briangoetz/lambda/collections-overview.html
[4] [Online]. Available: http://threadingbuildingblocks.org/
[5] T. A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference," in *PLDI*, 2004.
[6] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *TOPLAS*, 2006.
[7] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *PLDI*, 2006.
[8] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *ESEC/FSE*, 2007.
[9] R. Jhala and R. Majumdar, "Interprocedural analysis of asynchronous programs," *SIGPLAN Not.*, 2007.
[10] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," in *POPL*, 2007.
[11] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge, "Component-based lock allocation," in *PACT*, 2007.
[12] E. Bodden and K. Havelund, "Racer: effective race detection using aspectj," in *ISSTA*, 2008.
[13] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *ESEC/FSE*, 2009.
[14] M. Naik, P. Liang, and M. Sagiv, "Static Thread-Escape Analysis vis Dynamic Heap Abstractions," from Naik's website.
[15] P. Liang, O. Tripp, M. Naik, and M. Sagiv, "A dynamic evaluation of the precision of static heap abstractions," in *OOPSLA*, 2010.
[16] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for c," *ACM Trans. Program. Lang. Syst.*, 2011.
[17] D. Schonberg, "On-the-fly detection of access anomalies," *SIGPLAN Not.*, 1989.
[18] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," *SIGPLAN Not.*, 1990.
[19] J.-D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," *TOPLAS*, 1991.
[20] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *ICS*, 1991.
[21] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer, "Detecting data races on weak memory systems," *SIGARCH Comput. Archit. News*, 1991.
[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, 1997.
[23] M. Ronsse and K. De Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, 1999.
[24] M. Christiaens and K. De Bosschere, "Trade, a topological approach to on-the-fly race detection in java programs," in *JVM*, 2001.
[25] C. von Praun and T. R. Gross, "Object race detection," in *OOPSLA*, 2001.
[26] R. O'Callahan and J. Choi, "Hybrid dynamic data race detection," in *PPoPP*, 2003.
[27] H. Nishiyama, "Detecting data races using dynamic escape analysis based on read barrier," in *VM*, 2004.
[28] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: effective sampling for lightweight data-race detection," in *PLDI*, 2009.
[29] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *PLDI*, 2009.
[30] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "Racez: a lightweight and non-invasive race detection tool for production applications," in *ICSE*, 2011.
[31] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *PASTE*, 2001.
[32] [Online]. Available: http://jdk8.java.net
[33] Wala documentation. [Online]. Available: http://wala.sourceforge.net/
[34] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, 2005.
[35] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS*, 2009.
[36] 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*, 1999.
[37] Concurrency jsr-166 interest site - parallelarray. [Online]. Available: http://gee.cs.oswego.edu/dl/concurrency-interest/
[38] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," *SIGOPS Oper. Syst. Rev.*, 2005.
[39] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs," *Concurrency and Computation: Practice and Experience*, 2007.
[40] F. Chen, T. F. erbnu, and G. Rou, "jPredictor: a predictive runtime analysis tool for Java," in *ICSE*, 2008.
[41] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," in *PLDI*, 2006.
[42] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, 2003.
[43] D. Grossman, "Type-safe multithreading in cyclone," in *TLDI*, 2003.
[44] S. Qadeer and D. Wu, "Kiss: keep it simple and sequential," in *PLDI*, 2004.
[45] C. Flanagan and S. N. Freund, "Type-based race detection for java," in *PLDI*, 2000.
[46] C. Boyapati and M. Rinard, "A parameterized type system for race-free java programs," in *OOPSLA*, 2001.
[47] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," in *OOPSLA*, 2002.
[48] C. Flanagan and S. N. Freund, "Detecting race conditions in large programs," in *PASTE*, 2001.
[49] R. Agarwal and S. Stoller, "Type inference for parameterized race-free java," in *VMCAI*. Springer Berlin / Heidelberg, 2004.
[50] J. Rose, N. Swamy, and M. Hicks, "Dynamic inference of polymorphic lock types," *Science of Computer Programming*, 2005.

[51] C. Flanagan and S. N. Freund, "Type inference against races," *Sci. Comput. Program.*, 2007.
[52] C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *PLDI*, 2003.
[53] J.-D. Choi, A. Loginov, and V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," IBM Research Division, Thomas J. Watson Research Centre, Tech. Rep., 2001.