10th IEEE International Conference on Software Testing, Verification and Validation

Ariadne
Hybridizing Directed Model Checking and Static Analysis

Reed Milewicz
Sandia National Laboratories
Albuquerque, New Mexico 87123
Email: rmilewi@sandia.gov

Peter Pirkelbauer
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama 35223
Email: pirkelbauer@uab.edu

***Abstract*—While directed model checking has proven to be a powerful tool in the fight against concurrency bugs, scalability remains a concern due to the combinatorial explosion in size of the state space. Overcoming that combinatorial explosion requires the selection and/or parameterization of meta*-heuristics, but we are left with a persistent problem of having to provide or compute specialized knowledge of the program under consideration, and this limits the practical value of the technique. To circumvent that, this paper investigates directed model checking as a platform for the synthesis of results from other analyses. We introduce an open-source tool, Ariadne, which translates reports of suspected race conditions of a static analyzer (Petablox) to instrumentation using a source-to-source compiler (ROSE) that can be exploited by a model checker (Java Pathfinder). We detail the algorithm used, present experimental results, and outline directions for future research.**

***Index Terms*—model checking, heuristics, static analysis, race detection**

I. Introduction

One of the greatest obstacles to widespread adoption of directed model checkers is that they are heavy-weight tools that require significant human labor in order to render tractable results, as heuristics require design, selection, and parameterization. We call to mind Engler and Musuvathi 2004, a report on the use of model checkers that found that for the sake of scalability and ease of use, a model checker should "require as little input, annotations, and guidance as possible"[1]; if effective use of model checking technology proves too burdensome, programmers will forego its use. The need for human intervention is, by far, the most expensive form of invisible *technical debt*, that is, a resource cost that is deferred by the design of our bugfinding solution. To circumvent this problem, our work focuses on hybridizing model checking with static analysis. Oftentimes, a software development team will use multiple verification tools to test the correctness of their code, including analyses of source code, byte code, execution traces, comments, documentation, and formal specifications. Each of these analyses give us a window into the overall behavior of the program, and by incorporating these tools into the development lifecycle, the technical debt of relying on outside information has already been paid; by not taking advantage of this free credit, the model checker spends a significant amount of time rediscovering what has already been known.

On the other hand, if the foreign analysis could find the bugs efficiently and without fail, then we would have no need for the model checker; we expect our priors to either be imperfect (i.e. imprecise, unsound) or incomplete. In our case, as a complement to directed model checking, the soundness and scalability of static analysis are attractive, but we are forced to contend with its shortcomings. A survey by Mamun et al.[2], which compared the effectiveness of four different static analysis tools at finding Java concurrency bugs, showed that bugs were only detected 25.25% of the time on average. A separate, similar study of three tools by Kester et al.[3] measured a 31.62% detection rate. Moreover, true positives can be hidden amongst false positives: Mamun et al. found a 41.6% false positive rate, and Kester et al. 24.07%. Moreover, designing a scheme to translate that knowledge into a heuristic form is non-trivial. As a way of exploring this topic, and to build a foundation for future work, this paper introduces Ariadne, an algorithm and an experimental toolchain that hybridizes pure static analysis with directed model checking. Our contributions are as follows:

1) An algorithm for automatically mapping reports from a static analyzer of suspected race conditions to executable annotations using a source-to-source compiler.
2) A multi-objective heuristic function that leverages the dynamic annotations to inform the state space search.
3) A completely open-source toolchain that implements these algorithms.

The remainder of this paper is laid out as follows. In §II, we cover related work. In §III, we discuss the algorithms used by Ariadne and how they are realized. In §IV, we conduct experiments to demostrate the effectiveness of our approach in a rigorous way. Finally, in §V, we summarize our findings and outline a road map for future research.

The Ariadne analysis pipeline is built upon three different open-source frameworks: Petablox, ROSE, and Java Pathfinder. We shall introduce them here.

Petablox[4] is a program analysis platform for Java that originated with the doctoral work of Mayur Naik in 2008[5].

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.49

Petablox provides a dazzling variety of off-the-shelf static and dynamic analyses, including concurrency analyses for finding races, deadlocks, and atomicity violations.

ROSE[6], developed at the Lawrence Livermore National Laboratory (LLNL), is a source-to-source translation infrastructure for multiple languages, including C/C++, Fortran 77/95/2003, Java, and UPC.

Java Pathfinder is a fully configurable execution environment for Java bytecode programs developed by the NASA Ames Research Center and first released in 2005[7]. Java Pathfinder originated as a stateful, directed model checker but also supports numerous modes of execution and extensions for different kinds of analysis.

II. Related Work

There have been several works that have used static analysis to help prune the state space. Brat and Visser 2001[8] pair static analysis with JPF to perform on-the-fly partial order reduction. Visser et al. 2003[9] build on that earlier work and use static analysis to perform static slicing, partial evaluation, and partial order computation. Along these lines, research has been done to use model checking to isolate and explore portions of the input program in order to refine the results of a static analysis. Post et al. 2008[10] combine abstract interpretation (Polyspace) with the CMBC model checker to automatically eliminate false positives from the results of the static analyzer. More recent work in the vein of Post et al. can be seen in Darke et al. 2012[11] and Muske and Khedker 2015[12]. Another avenue of research has been to develop a collaborative framework for static analysis and model checking, so that we can make trade-offs between the efficiency of static analysis and the precision of model checking. Beyer et al. 2007[13] use the BLAST model checker to perform composite analysis alongside shape analysis and pointer analysis. Chen and MacDonald 2008[14] demonstrate a bi-directional scheme for collaboration between a static analysis tool (Soot) and a model checker (JPF).

These works overlap with ours in their premises, but differ from ours in their methodologies and goals. Like Brat and Visser 2001 or Visser et al. 2003, we aim to use static analysis to improve upon the capabilities of model checking, but our work focuses on producing more informed heuristics rather than on reducing the size of the state space. Likewise, as with works such as Post et al. 2008, we assume that the results of the static analyzer cannot be fully trusted due to the presence of false positives, but unlike those works, our goal is to refine the performance of the model checker rather than the static analyzer. Work analogous to ours has been done to merge model checking with dynamic analysis. Groce and Joshi 2008[15] use source-to-source instrumentation using the CIL framework to add instrumentation to extend the functionality of the SPIN model checker with run-time checks. More recently, Milewicz and Pirkelbauer 2016[16], illustrates an approach for combining model checking with execution trace mining to inform heuristic search.

III. Implementation

We shall provide an overview of Ariadne. First, a concurrent program P is passed to Petablox, which performs static analyses and highlights potential race conditions. Ariadne extracts from the bug reports the paths through the call graph of P that lead to the suspected violations. Ariadne then passes P to ROSE, which parses the input program to produce an internal intermediate representation (IR), including a detailed abstract syntax tree (AST). Ariadne instruments race-relevant code paths with callable annotations; the AST is then unparsed by ROSE, giving us our instrumented program P'.

We then pass P' to JPF for in-depth analysis. During runtime, calls to annotations allow Ariadne to track the progress of different threads through the code. Using this information, our heuristic function encourages JPF to schedule threads which are most likely to cause a suspected race condition when running in concert.

In this section, we describe each of these steps in depth.

A. Performing Static Analysis using Petablox

For two read/write instructions $inst_i, inst_j \in I$ where at least one is a write operation, a race condition is declared by Petablox if the following conditions are met:

1) $inst_i$ reachable from the thread-root of a thread t_a, and $inst_j$ is reachable from the thread-root of a thread t_b. To determine this, Petablox performs call graph analysis.
2) $inst_i$ and $inst_j$ can access the same memory location shared by t_a and t_b. This is decided by may-alias analysis and thread-escape analysis.
3) t_a and t_b are able to execute $inst_i$ and $inst_j$ in parallel. Petablox determines this through may-happen-in-parallel analysis. Along with this, Petablox also performs conditional must-not alias analysis to confirm that t_a and t_b are not prohibited from executing the instructions in parallel due to any common locks.

As a guiding example, in Fig. III-A we present a simple race condition involving a shared global variable. The client class `A` instantiates threads (not shown) that call methods `A.foo` and `A.bar`, which call a library class `B`. The use of the `synchronized` keyword on `B.baz` ensures that a happens-before relationship exists between any concurrent calls to that method, but they do not order concurrent calls to other, non-synchronized methods. Because threads in `A` call `B.baz` and `B.qux` concurrently, it is possible to have unsynchronized updates to the shared variable `B.x`.

For our purposes, a race condition is represented as $r_{id} = (lhs_{id}, rhs_{id})$, a 2-tuple where $lhs_{id} = linstr_1 ... linstr_{\|lhs_{id}\|}$ and $rhs_{id} = rinstr_1 ... rinstr_{\|rhs_{id}\|}$ are lists of instructions that lead to a conflicting pair of accesses to shared memory, which we will refer to as the left-handed and right-handed paths respectively. We note that this distinction between the two is arbitrary and is merely used here as an explanatory aid; (lhs_{id}, rhs_{id}) and (rhs_{id}, lhs_{id}) would represent the exact same race. Parsing each race condition report produced by Petablox gives us our set of race conditions $R = r_1 ... r_{\|R\|}$.

Listing 1. class A (Client)

```
public static void foo(){B.baz(false)}
public static void bar(){B.qux()}
```

Listing 2. class B (Library)

```
private static int x = 0;
public static synchronized void baz(boolean flag){
        if(flag)
                x++;
        else
                B.norf();
}
private static void norf() { B.baz(true) };
public static qux(){x−−;}
```

Listing 3. Petablox Bug Report

```
Possible race condition detected!
Path of abstract thread #1:
        A.foo() in Thread.run()
        B.baz(false) in A.foo
        x++ in B.baz
Path of abstract thread #2:
        A.bar() in Thread.run()
        B.qux() in A.bar
        x−− in B.qux
```

Fig. 1. A contrived example of a race condition between client threads calling into a common library. Concurrent calls to `A.foo` and `A.bar` can result in a race condition due to improper synchronization over accesses to `B.x`. Note that because the analysis is context-insensitive, Petablox reports the shortest path to the violation irrespective of inputs (e.g. `x++` is only reachable through `B.norf`.)

B. Converting Petablox Reports to Annotations

To encode race condition information, for each scope containing code relevant to any race condition, we compute two relevancy matrices $M_{enter}, M_{exit} \in \mathbb{R}^{2*\|R\|x2*\|R\|}$. A relevancy matrix is an incidence matrix defined over the set of all paths reported to Petablox that lead to race conditions. The intent of M_{enter} and M_{exit} is two-fold. First, for a thread entering or exiting an annotated scope, they estimate the "distance" of that thread to code that is suspected to cause a race condition, and second, they communicate the relevance of that thread executing that code to threads following opposite-handed paths towards suspicious code (with the understanding that we need both threads to reach their respective targets in order to produce the violation). We shall now explain how these matrices are composed.

For a given scope $b \in V_{blocks}$, for each individual race $(lhs_i, rhs_i) \in R$, we compute a block matrix B_i^b of the form

$$B_i^b = \begin{array}{c} \\ lhs \\ rhs \end{array} \begin{array}{c} \begin{array}{cc} lhs & \quad rhs \end{array} \\ \begin{bmatrix} 0 & \mathrm{v}(b, lhs_0) \\ \mathrm{v}(b, rhs_0) & 0 \end{bmatrix} \end{array} \tag{1}$$

which encodes the relevance of a thread being in b with respect to race condition r_i, with $B^b_{i\,lhs,rhs}$ being the relative position of a thread in b on the path lhs and its relevance to a thread on path rhs, and $B^b_{i\,rhs,lhs}$ being the relative position of a thread in b on the path rhs and its relevance to a thread on path lhs. The function v is our value function, and is defined as

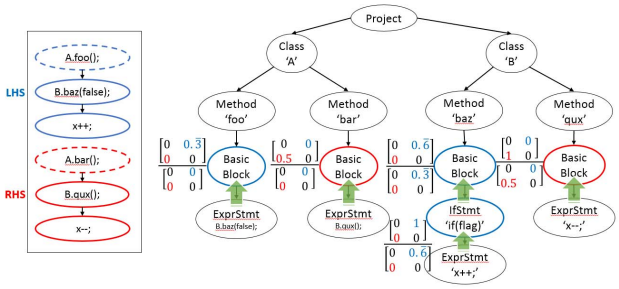

Fig. 2. A visual diagram describing the instrumentation process as applied to our example program. Statements in the paths belonging to the race are mapped to statement nodes in the AST. Ariadne traverses back up the tree to find the scopes enclosing the statement, and calculates the relevancy matrices using the process described in §III-B.

$$\mathrm{v}(b, \mathrm{path}) = \begin{cases} \frac{\mathrm{path.index}(b)}{\mathrm{path.length}} & \text{if } b \in \mathrm{path} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For example, for a race condition r_i that involves the scope b, $B_i^b = \left(\begin{smallmatrix} 0 & 0.25 \\ 0.75 & 0 \end{smallmatrix}\right)$ means that a thread currently in scope b is 25% of the way towards reaching the end of the left-handed path of r_i and 75% of the way towards reaching the end of the right-handed path of r_i. Note that this is a rough estimate because the underlying analysis is context-insensitive; we do not know the true distance.

Next, by taking the direct sum of these block matrices, we get

$$M^b_{race} = \oplus_{i=0}^{\|R\|} B_i^b = \begin{array}{c} \\ lhs_0 \\ rhs_0 \\ \vdots \\ lhs_{\|R\|} \\ rhs_{\|R\|} \end{array} \begin{array}{c} \begin{array}{cccccc} lhs_0 & rhs_0 & \cdots & lhs_{\|R\|} & rhs_{\|R\|} \end{array} \\ \begin{bmatrix} 0 & \mathrm{v}(b, lhs_0) & & & 0 \\ \mathrm{v}(b, rhs_0) & 0 & & & \\ & & \ddots & & \\ & & & 0 & \mathrm{v}(b, lhs_{\|R\|}) \\ 0 & & & \mathrm{v}(b, rhs_{\|R\|}) & 0 \end{bmatrix} \end{array} \tag{3}$$

which is our relevancy matrix for the scope. Note that because the matrix is tridiagonal and the leading diagonal is all zeroes, in the actual implementation the relevancy matrix can be stored as two 1D-arrays of length $\|R\|$.

C. Instrumentation of Source Code

After receiving the bug report from Petablox and parsing the XML documents to extract the race condition records $R = r_1 ... r_{\|R\|}$, Ariadne parses the source code of P using ROSE to get the AST. The lists of statements belonging to each race are mapped to corresponding statement nodes within the AST. To allow for selective instrumentation, statements with missing statement nodes are omitted. For example, we may perform static analysis on a library A that calls a library B, and if we detect potential bugs in B alone, we may want to craft our own test harness to analyze B exclusively. If we only pass B to ROSE, then path elements related to A are automatically excised.

Next, Ariadne traverses up from each statement node to get the enclosing scopes. For each scope, we compute the relevancy matrices and attach the annotation records to that

Listing 4. class A (Client)

```
public static void foo(){
        float[] rmatrix_foo_enter = {{0.333},{0.0}};
        Ariadne.annotate(rmatrix_foo_enter);
        B.baz(false);
        float[] rmatrix_foo_exit = {{0.0},{0.0}};
        Ariadne.annotate(rmatrix_foo_exit);
}
public static void bar(){
        float[] rmatrix_bar_enter = {{0.0},{0.5}};
        Ariadne.annotate(rmatrix_bar_enter);
        B.qux()
        float[] rmatrix_bar_exit = {{0.0},{0.0}};
        Ariadne.annotate(rmatrix_bar_exit);
}
```

Listing 5. class B (Library)

```
private static int x = 0;
public static synchronized void baz(boolean flag){
        float[] rmatrix_baz0_enter = {{0.666},{0.0}};
        Ariadne.annotate(rmatrix_baz0_enter);
        if(flag) {
                float[] rmatrix_baz1_enter = {{1.0},{0.0}};
                Ariadne.annotate(rmatrix_baz1_enter);
                x++;
                float[] rmatrix_baz1_exit = {{0.5},{0.0}};
                Ariadne.annotate(rmatrix_baz1_exit);
        } else
                B.norf();
        float[] rmatrix_baz0_exit = {{0.333},{0.0}};
        Ariadne.annotate(rmatrix_baz0_exit);
}
private static void norf() { B.baz(true) };
public static void qux(){
        float[] rmatrix_qux_enter = {{0.0},{1.0}};
        Ariadne.annotate(rmatrix_qux_enter);
        x−−;
        float[] rmatrix_qux_exit = {{0.0},{0.5}};
        Ariadne.annotate(rmatrix_qux_exit);
}
```

Fig. 3. Our example program post-instrumentation.

node. This process is illustrated in Fig. 2 as it applies to the example code we provided in Fig. III-A.

Ariadne then makes a final pass over the AST to instrument all labeled scopes. To each scope, we prepend and append (1) a uniquely-named, statically-declared array variable that holds the relevancy matrix and (2) a call to a custom library method `Ariadne.annotate()` to which we pass the array variable. Normally, this means inserting the instrumentation at the start and end of a basic block, but if the race-relevant code occurs within a branch statement (e.g. the conditional of a `for` loop), the instrumentation is instead inserted just before and after the branch. Ariadne then concludes this process by adding an import statement for Ariadne's run-time library. Finally, the AST is unparsed by ROSE to give us our instrumented program P' which is now ready to be analyzed with JPF (see Fig. III-C).

D. Handling of Dynamic Annotations

Calls to `Ariadne.annotate()` are lightweight dynamic annotations that announce the caller's relative position in the program. The body of the `annotate` is in fact empty; the responsibility for tracking these calls falls to JPF. JPF provides an Observer pattern implementation that allows users to attach listeners to the search routine and to the JVM to observe and interact with the model checker during execution. Many important routines, such as race detection and deadlock detection, are implemented as listeners. To track annotation calls, Ariadne attaches an `AriadneListener` instance to JPF prior to execution. This listener performs the following functions:

1) Whenever a thread is created, the listener attaches an annotation to the `ThreadInfo` object associated with that thread in the current state. This annotation will hold the last known relevancy matrix for that thread (initially a zero matrix). The annotation is made inheritable such that the thread will carry the annotation with it as JPF generates and stores subsequent states.
2) Whenever a thread calls the `Ariadne.annotate()` method, the listener intercepts the call and captures the argument, a new relevancy matrix. The annotation associated with the thread is updated to contain this new relevancy matrix.

Annotating the threads directly, as opposed to storing the annotations in a separate metadata facility, is advantageous for two reasons. First, always having the most recent relevancy data on hand means that it can be fetched in constant time. Second, whenever we need to backtrack to a previous state, there is no need to recalculate or roll back the annotation history for a thread. This solution does add to the memory overhead, but only by a fixed $2 * \|R\| * N_{threads} * 32$ bits per state (assuming single precision arithmetic is used) for storing the relevancy matrix.

E. Defining the Heuristic

Assume that at timestep i we reach a state S_i from the start state S_0 through transitions $Tr_0..Tr_i$, and we have a series of unexplored transitions $Tr_0^{new}..Tr_N^{new}$ to choose from, with one transition for each of our currently active threads $t_0..t_N$. To control the cost of performing our heuristic, we borrow an idea from Groce and Visser 2002[17] and set a window of size $limit$ on the path, giving us a set of transitions $Tr_{i-limit}..Tr_i$, which we call our history.

For an arbitrary candidate transition Tr_{next} and its associated thread t_{next}, we have a relevancy matrix M_{next} stored in S_i. To measure the relevance of t_{next} to threads previously scheduled in the history, we want to compare their relevancy matrices $M_{i-limit}..M_{i-1}$ against the candidate. For example, if M_k has a high value for the entry lhs_0, then t_k would want t_{next} to have a high value for rhs_0 in M_{next}.

To do this, we first sum together all of the matrices of the preceding threads. We will call this result M_{pred}. We then take the Hadamard product of M_{pred}^{T} and M_{next}. This means that lhs values of M_{next} are multiplied by the rhs values of M_{pred}, and vice versa; this gives us our result, $M_{priority}$. Finally, we extract and sum the diagonals to form a vector of length $\|R\|$. That is,

$$H_{priority}(\text{path}, \text{limit}) = \langle lhs_i + rhs_i : lhs_i, rhs_i \in M_{priority} \rangle \quad (4)$$

$$M_{priority} = M_{pred}^{\mathrm{T}} \circ M_{next} \tag{5}$$

$$M_{pred} = \left(\sum_{i=\text{path.length}-\text{limit}}^{\text{path.length}} \text{path.get}(i).\text{getRMatrix}() \right) \tag{6}$$

$$M_{next} = \text{path.get}(path.length - 1).\text{getRMatrix}() \tag{7}$$

Each element in the output vector is a measure of the preference of recently scheduled threads for the candidate t_{next} according to the codepath(s) that the thread is on. In other words, each element represents an independent heuristic that measures proximity to a reported race condition. To combine these heuristics, we can employ a multiobjective search strategy, which is an extension of scalar graph search to support vector-valued costs[18]. We divide the search problem into multiple sub-problems that can be solved independently; we introduce a *partition function* that splits a vector input into multiple sub-vectors, each of which can serve as the basis for a separate search. These instances can be run sequentially, in parallel on the same machine, or distributed across a cluster. For this work, we employ static (non-communicating) partitioning, a topic which has been previously explored in Staats and Păsăreanu 2010[19].

IV. Evaluation

In our preliminary experiments, we found that Ariadne delivered optimal or near-optimal results when tested on commonly used concurrency bug benchmarks. However, tempting as it was to claim improvements by up to a factor of 180, we concluded that these results were not a fair representation of our algorithm. The majority of these benchmarks are *concurrency error kernels*, deliberately simple programs that express quintessential varieties of concurrency bugs. With respect to our approach, these benchmarks are mercifully easy: there is usually only one code pair that can cause a race condition, the programs are small enough that there is little else that could trigger false positives, and both the inputs and the control structure of the programs are relatively simple.

For these reasons, we turn our attention to *de novo* concurrent program generation. Notable recent research on this topic is that of Steffen et al. 2014[20], who have provided a practical model for generating concurrent benchmarks based on LTL synthesis and translation of labeled transition systems (LTS) to equivalent code. In a similar vein, we developed a ROSE-based tool for randomly generating concurrent Java programs that contain race conditions.

Our program generation method uses the Boost Graph Library (BGL)[21] to construct a graph which we treat as an LTS. A node translates to a method in our concurrent library, and a directed edge indicates that the method calls another method. In order to simulate input and control flow non-determinism, we make use of JPF's Verify library to non-deterministically choose which edge a thread will take.

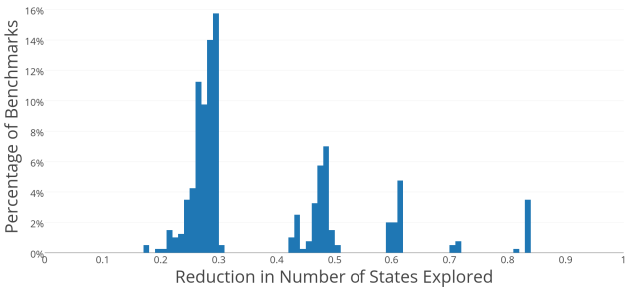

Fig. 4. Histogram of the results of the branching factor experiment across all 400 benchmarks tested. The lower bound on the improvement of Ariadne over BFS was a 17.89% reduction in states explored, and the upper bound was 83.81%, with a median of 27.75%.

To this graph, we add labels to nodes that represent read and write methods on shared memory. These methods can either be unguarded or guarded by locks whose identifiers are obfuscated. Petablox flags both the true and false positive cases as potential race conditions, and Ariadne supplies us with paths through the graph to reach those violations.

Our experiments were performed on an Intel 20-core Phi with 130 GB of RAM running Java 7, with the most recent version of Java Pathfinder.

In our first set of experiments, we examine the sensitivity of our approach to increasing complexity of the control flow. To do this, we generate perfect n-ary trees of fixed depth (3) and we vary the branching factor. We embed a single unsynchronized read and write pair at two randomly chosen leaves. In our benchmark, we dispatch a fixed number of threads (4) at the root. We compare the search performance of A* search plus Ariadne vs. an uninformed, breadth-first search as the branching factor increases from 3 to 6, by testing on 100 benchmarks each time. For this experiment, we measure the total number of states explored to find a bug (as a proxy for time/space costs).

For our second set of experiments, we consider the interplay of having many different race conditions and different mixtures of true and false positives. We consider a tree of fixed depth (6), branching factor (2), and number of threads (4). We introduce three false positives and one true positive, and we consider the effects of partitioned searches from size 2 to 4. We do this to simulate a partitioned, multiobjective search. For example, Petablox may hint at some 60 different race conditions, and we may not have the resources needed to carry out 60 different parallel searches. However, through static partitioning, we can reduce that number to 20 searches with partition size three or 15 searches with partition size four, with partitions ordered lexicographically.

A. Results

We begin by examining the results of the synthetic benchmark tests. First, we look at the branching factor tests. Since relevancy information is only updated when a thread reaches an annotated block of code, there can be many intervening unlabeled blocks that "inherit" the relevancy values of the

last annotation. In these benchmarks, a thread executing any descendant of an annotated interior node is given at least as much priority as a thread entering that annotated node, even though only a fraction of those descendants are actually on the path to a race. The concern then is that the information we receive from the static analyzer may be too imprecise to be helpful in our search. While this is guaranteed to be problematic for greedy search, we find that this concern is over-stated for Ariadne under A* search.

On average, Ariadne and BFS find the bug in the 3-ary case after exploring 1650 and 2561 states respectively, 3586 and 5750 in the 4-ary case, 6784 and 11083 in the 5-ary case, and 11915 and 19067 in the 6-ary case. As can be seen in Fig. 4, the median improvement in performance held steady at around 27.75% (an average reduction by a factor of $\frac{1}{e}$). These results were independent of our choice of branching factor, and the distribution of performance gains is virtually identical in all cases. In terms of time and space performance, this translated to a 54% reduction in execution time with negligible space overhead introduced by the recording of annotations (e.g. 20 megabytes out of 2 gigabytes).

Next we consider the partitioning experiments. On the benchmarks we tested, the average partitioned search involving the sole true positive found the bug 11.12% slower for partitions of size 2, 7.72% slower for partitions of size 3, and 9.16% slower across all partitions of size 4. That is, we witnessed a slightly lower overhead when we increased the partition size, even though an increase in partition size means that, in the average case, we are less likely to pursue the true positive early. False positives slow the search, but we are refunded to the extent that false positive paths overlap with true positive paths.

V. Conclusion and Future Work

In this work, we have introduced Ariadne, an algorithm and toolchain for improving race condition detection by directed model checking via static analysis. Ariadne uses ROSE to translate race condition reports from Petablox to dynamic metadata annotations that guide Java Pathfinder towards suspected violations. The central challenge is that while the static analysis is inexpensive, it can detect an abundance of potential race conditions, many of which can be misleading false positives. Our results have shown that through a combination of multiobjective and parallel search, Java Pathfinder can leverage this information at scale, delivering significant improvements in performance. Moreover, Ariadne is merely the beginning: the approach we have proposed can be readily extended to incorporate arbitrarily many different analyses. We believe that using directed model checking as a platform for unifying analyses will make it substantially easier for developers to understand and validate the semantics of concurrent code.

VI. Acknowledgements

This project is supported by the National Science Foundation under grants CNS-0821497 and CNS-1229282.

References

[1] D. Engler and M. Musuvathi, "Static analysis versus software model checking for bug finding," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 191–210.

[2] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for java concurrency bugs," in *Third Swedish Workshop on Multi-Core Computing (MCC-10). Chalmers University of Technology*, 2010.

[3] D. Kester, M. Mwebesa, and J. S. Bradbury, "How good is static analysis at finding concurrency bugs?" in *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*. IEEE, 2010, pp. 115–124.

[4] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *23rd Symposium on Foundations of Software Engineering*, 2015.

[5] M. Naik, A. Aiken, and J. Whaley, *Effective static race detection for Java*. ACM, 2008, vol. 41, no. 6.

[6] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[7] W. Visser, C. S. Psreanu, and S. Khurshid, "Test input generation with java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[8] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE, 2001, pp. 262–269.

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

[10] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 188–197.

[11] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise analysis of large industry code," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1. IEEE, 2012, pp. 306–309.

[12] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 270–280.

[13] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification*. Springer, 2007, pp. 504–518.

[14] J. Chen and S. MacDonald, "Towards a better collaboration of static and dynamic analyses for testing concurrent programs," in *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. ACM, 2008, p. 8.

[15] A. Groce and R. Joshi, "Extending model checking with dynamic analysis," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2008, pp. 142–156.

[16] R. Milewicz and P. Pirkelbauer, "Refinement of structural heuristics for model checking of concurrent programs through data mining," *Computer Languages, Systems & Structures*, vol. 45, 2016.

[17] A. Groce and W. Visser, "Model checking java programs using structural heuristics," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 12–21.

[18] B. S. Stewart and C. C. White III, "Multiobjective a*," *Journal of the ACM (JACM)*, vol. 38, no. 4, pp. 775–814, 1991.

[19] M. Staats and C. Psreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 183–194.

[20] B. Steffen, M. Isberner, S. Naujokat, T. Margaria, and M. Geske, "Property-driven benchmark generation: synthesizing programs of realistic structure," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 5, pp. 465–479, 2014.

[21] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.