Detecting Data Races with Java PathFinder[*]

Sergey Kulikov[1**], Nastaran Shafiei[1], Franck van Breugel[1], and Willem Visser[2]

[1] DisCoVeri Group, Department of Computer Science and Engineering
York University, 4700 Keele Street, Toronto, M3J 1P3, Canada

[2] Department of Mathematical Sciences, Computer Science Division
University of Stellenbosch, Private Bag X1, 7602 Matieland, South Africa

1 Introduction

Roughly speaking, a (data) race on a shared variable arises in a concurrent program if two threads access that variable simultaneously and the accesses are conflicting, that is, at least one of them writes to the variable. Although some races are benign, races often are an indication of bugs. Hence, tools that detect them are invaluable to those writing concurrent programs.

Many tools have been developed to detect races. These tools are based on two types of race detection techniques: dynamic and static. In dynamic race detection, a single execution of a concurrent program is checked for races. One of the key approaches to detect races dynamically is based on locksets and has been popularized by the Eraser tool [1].

In this paper, we focus on static race detection. All potential executions are considered in static race detection. Although this approach gives rise to tools that are usually sound (that is, the races that are reported by the tool are real races), the tools are generally not complete (that is, not all races are always reported). Several different approaches exist to statically detect races. Here, we concentrate on model checking. In [2] model checking is exploited to detect races in programs written in an extension of C. Here, we focus on Java PathFinder (JPF)[3] [3]. This is a model checker for Java bytecode. It has been developed in such a way that it can easily be extended. Extensions to detect races is the topic of this paper.

The lockset algorithm and its numerous variations are usually exploited for dynamic race detection. However, this algorithm has also been used for static race detection. A variation on the lockset algorithm has been implemented in JPF.[4] In this paper, we propose a different way to use JPF to detect races.

2 The New Race Detector

JPF explores all potential executions in a systematic way. Each execution is a sequence of transitions. Each transition takes the system from one state to

[*] This research is supported by NSERC.

[**] Current affiliation: MKS, 410 Albert Street, Waterloo, N2L 3V3, Canada

[3] `http://babelfish.arc.nasa.gov/trac/jpf/`

[4] The class `gov.nasa.jpf.tools.RaceDetector` contains this implementation.

another. Each transition consists of a sequence of bytecode instructions. JPF groups bytecode instructions such that an instruction that manipulates a shared variable is the first one of a transition. We exploit this fact in our race detector.

The idea behind our race detector is fairly simple. In every state that JPF visits, we check all actions that can be performed next. If this collection of actions contains at least two conflicting accesses of a shared variable, then a race on the shared variable is reported. A similar approach in a considerably simpler setting has been proposed in [4].

To prove that our approach is sound, we consider a simplified version of the Java memory model presented in [5]. The happens-before relation is defined as the transitive closure of the program order and the synchronization order. The program order captures the order in which the actions occur within a thread. The synchronization order captures the orders in which synchronization actions within different threads may occur. Two actions are concurrent if either one does not happen before the other. For the details, we refer the reader to [5, Section 2.1]. Now we are in a position to formally state that our algorithm is sound.

Theorem 1. *Let a and b be either a read or a write action. There exists an execution in which a and b are concurrent if and only if there exists a state in which a and b are enabled.*

Hence, there are concurrent and conflicting accesses of a shared variable (that is, a race on the shared variable) if and only if there exists a state in which those conflicting accesses are enabled. The latter is checked by our race detector.

3 Partial Order Reduction

One of the biggest challenges of JPF, as well as other model checkers, is the notorious state explosion problem. One of the approaches to battle the state explosion problem is partial order reduction (POR). JPF is also applying a POR technique to cut down the state space. Basically by putting POR in effect, JPF combines more bytecode instructions into a single transition. This invalidates Theorem 1 and, hence, our race detector is not sound anymore. However, we have modified JPF's POR so that one can partially enable POR. In that case, we maintain soundness, but the state space is generally larger than when POR is fully enabled. In the table below, we present the size of the state space for a number of concurrent Java programs when POR is fully, partially and not enabled. Note that partially enabling POR only mildly increases the size of the state space and maintains soundness. Not enabling POR also maintains soundness but increases the size of the state space much more.

	Producer Consumer	Reader Writer	Sleeping Barber	Cigarette Smokers
fully	329,506	1,618,672	222,337	298,493
partially	492,588	1,930,804	294,853	306,007
not	1,307,178	5,348,875	1,368,784	559,159

4 Comparison with the Old Race Detector

Let us now compare our race detector with JPF's original race detector. As we have shown above, our race detector is sound. JPF's original race detector is however not sound. The key idea behind the lockset algorithm, on which the original race detector is based, is the following implication. If for a shared variable v there exists at least one lock ℓ such that ℓ is held during all conflicting accesses of v in an execution, then the execution is free of races on v. However, locking is not the only programming idiom that can be used to prevent races. Instead, for example, semaphores can be exploited. Assume that a `semaphore` and a `variable` are shared by two threads and the `semaphore` is initialized to 1. Assume also that both threads execute the following code snippet.

```
  semaphore.acquire(); variable++; semaphore.release();
```

In this case, no locks are held when the `variable` is accessed. As a consequence, JPF's original race detector reports a potential race on `variable`. This is however not a real race, as is confirmed by our race detector which does not report a race.

JPF has a number of search algorithms to traverse the state space such as depth-first search (DFS) and breadth-first search (BFS). The user can configure JPF to use any of these algorithms. But JPF's original race detector has been built on the assumption that JPF uses DFS. Assume that two threads share a `variable` and that both threads increment the `variable`. If JPF uses DFS to traverse the state space, the original race detector reports a race on `variable`. However, if JPF uses BFS instead, then the original race detector does not report the race. Also for other traversal algorithms, the implementation of the original race detector needs to be changed. In contrast, our race detector is independent of the search algorithm of JPF and, hence, can be used with any search algorithm without any modification.

In our race detector, we have also addressed some of the limitations of JPF's original race detector. Due to lack of space, we will not discuss those here.

To evaluate the performance of our race detector, we ran JPF on a set of concurrent Java programs. For each program, JPF was run a hundred times with three different settings. The table below shows the average running time in milliseconds. The first row contains the number of lines of code for each program. The second row is obtained using the default setting of JPF. The third and fourth row report the results with the old race detector and the new one enabled, respectively. The fifth row compares the overhead of the old and the new race detector.[5] It shows that the overhead of both race detectors is very small. It also shows that our race detector is more efficient than JPF's original race detector.

[5] overhead ratio = $\frac{\text{old} - \text{default}}{\text{new} - \text{default}}$.

	Producer Consumer	Reader Writer	Sleeping Barber	Cigarette Smokers
size	86	103	153	139
default	44791	41039	55082	13224
old	44888	42572	64617	13941
new	44805	41116	62949	13618
overhead ratio	6.9	19.9	1.2	1.8

5 Conclusion

If one uses JPF to verify some concurrent Java code, one may as well switch on JPF's race detector, since its overhead is very small as we have shown.[6] We believe that our new race detector is superior to JPF's original race detector, since the former is sound whereas the latter is not, the former addresses some of the limitations of the latter, and the former is generally more efficient than the latter. Our modification of JPF's POR allows us to enable POR partially. This increases the state space only slightly (in comparison with POR fully enabled) and ensures that our race detector remains sound (in contrast to POR fully enabled). Our race detector has already been used in other research. For example, in [6] it is used in the context of self healing of races.

References

1. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15**(4) (November 1997) 391–411
2. Henzinger, T., Jhala, R., Majumdar, R.: Race checking by context inference. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington, DC, USA, ACM (June 2004) 1–13
3. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2) (April 2003) 203–232
4. Pollack, K.: Extending IMP to support threads with race detection by model checking. Unpublished (2004)
5. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Chicago, IL, USA, ACM (June 2005) 378–391
6. Hrubá, V., Křena, B., Vojnar, T.: Self-healing assurance based on bounded model checking. In: Proceedings of the 12th International Conference on Computer Aided Systems Theory. Volume 5717 of Lecture Notes in Computer Science, Las Palmas de Gran Canaria, Spain, Springer-Verlag (February 2009) 295–303

[6] Note that we do *not* claim that JPF is the best tool to use to detect races. Numerous special purpose tools are superior to JPF for detecting races.