SRD: Static Data Race Detection for Concurrent Programs

Yang Zhang, Yanan Liang, and Dongwen Zhang*

School of Information Science and Technology, Hebei University of Science and Technology, Shijiazhuang, 050000, China

Abstract

Data race probably occurs when many threads concurrently access the same memory location and at least one is a write thread. Data race detection suffers from false negatives and false positives. How to detect data race and avoid false negatives and false positives has become a hot topic. This paper proposes a static data race detection methodology to eliminate false negatives and false positives. We use Soot to conduct intra-thread and inter-thread analysis. Our data race detection focuses on variable access events that are collected from call graphs. Several program analysis technologies, such as alias variable analysis, alias lock analysis, happens-before analysis, constraint graph, and slicing analysis, are used to improve the coverage and precision of the detection results. In the experimentation, several benchmarks, such as raytracer, sor, and mergesort, have been selected to evaluate our methodology. Experimental results show that SRD can not only eliminate fake races but also identify potential races. Furthermore, SRD can detect more positive races than the existing tool RVPredict.

Keywords: data race; intra/inter-thread analysis; alias analysis; happens-before; concurrent program

(Submitted on August 18, 2018; Revised on September 16, 2018; Accepted on October 12, 2018)

© 2018 Totem Publisher, Inc. All rights reserved.

1. Introduction

The prevalence of multi-core processors has increased the use of concurrent programs. However, concurrent programs suffer from data race. Data race means that two or more threads access the same memory location without timing constraints and at least one of these access operations is write in a multithreaded program [1]. It often leads to non-deadlocked concurrent defects [2] and accounts for a large proportion in all concurrent defects.

Race detection can be used in static analysis [3-7] and dynamic analysis [8-10]. Dynamic analysis obtains accurate information of variables and aliases through program instrumentation. However, there is a large amount of interleaving space of executing and large detection overhead due to the uncertainty of thread scheduling. The dynamic detection is not comprehensive and often has many false negatives. Compared with dynamic analysis, static analysis can detect the data race quickly and comprehensively. However, static detection is only an incomplete approximation algorithm due to the indeterminacy of static analysis [11].

To improve the accuracy of data race detection, this paper proposes a static data race detection methodology and tool for multithreaded concurrent programs to decrease false negatives and false positives. Our methodology converts the source code into the Soot Intermediate Representation (IR) based on the Soot analysis framework [12-13] and then injects the analysis into the Soot scheduling process. The intra-thread analysis stage mainly presents the analysis of intra-thread function call graphs and variable access events within thread, and this stage can cover all variable accesses in each thread. In the inter-thread analysis, the potential data race can be found by alias variable analysis, which improves the coverage of the race detection. The fake race will be removed via alias lock analysis and happens-before analysis to improve the accuracy of race detection.

This paper is organized as follows. Section 2 introduces the motivation using an example and describes the advantages and disadvantages of existing detection algorithms. Section 3 presents the race detection methodology. Section 4 presents

* Corresponding author.

E-mail address: zdwwtx@hebust.edu.cn
the experimental results. Related works are examined in Section 5. Finally, conclusions are drawn in Section 6.

2. Motivation

In this section, we will illustrate how race conditions happen and present some disadvantages of existing detection through an example. In Figure 1, two threads are created and start to run at the same time. The access to the variable \( y \) is not protected by the monitor \( m \) in thread \( t1 \). It may be executed in parallel with the access to the variable \( y \) in thread \( t2 \). In other words, it is probable that the variable \( y \) is written by two threads at the same time. As a result, the data race happens in lines 8 and 11.

Many works introduce happens-before (HB) [14] to determine the data race according to the sequence between two events. However, data race could be false negatives by considering of thread interleaving. Considering two execution paths \( p1 = [\text{main}, t1, t2] \) and \( p2 = [\text{main}, t2, t1] \) in Figure 1, the code in lines 8 and 11 can be executed in parallel in the path \( p1 \), there is no ordering, and the race can be detected. However, thread \( t2 \) acquires \( lock(m) \) before thread \( t1 \) in the path \( p2 \), so accessing the variable \( y \) in line 11 is executed with protection and the variable \( y \) cannot be accessed in line 8 at the same time. The existence of ordering in the path \( p2 \) will be a false negative.

The execution of the lockset-based [15] methods is so strict that it avoids many false negatives. Although lockset-based detection is immune to thread interleaving, it leads to false positives. In multithreaded programs, locks are applied to protect critical sections, in which accesses to shared variables occur. If a shared variable is not protected by any locks during the executed processes of programs, data race may occur on this variable. In Figure 1, the lock sets in lines 1 and 6 computed by lockset-based methods are \( \text{null} \) and \( \{m\} \), respectively. The intersection of the two lock sets is \( \text{null} \), which determines the existence of data race between the two lines. In fact, they cannot access variable \( x \) at the same time because the statement in line 1 is executed before the thread \( t1 \) is created. The lockset-based data race detection between the two lines is a false positive.

Considering the advantages and disadvantages of lock-set and happens-before, hybrid methods, such as ThreadSanitizer [16], Acculock [17], and SimpleLock+ [18], are proposed. Our approach not only refers to lockset but also considers alias locks. Timing constraint graph and program slicing are applied to happens-before analysis.

3. Race Detection

SRD uses both intra-thread and inter-thread analysis to detect the data race through the program analysis tool Soot. Soot starts from constructing the call graph by some methods in class \( \text{ReachableMethods} \) and \( \text{TransitiveTargets} \) provided by the Soot call graph. The extension mechanism Pack (jtp, stp etc.) includes several transformations to get customized call graphs. The customized call graphs can be added to the Soot scheduling as a new transformation to analyze the domain-specific programs. We preserve those call graphs that variable accesses occur in the called function and then collect the information of access events. For inter-thread analysis, pre-race can be detected via contrasting detection between threads, which can find all possible races. Alias analysis and happens-before analysis are used to improve the coverage and precision of the detection. Alias variable analysis checks the consistency of variable memory location, while alias lock analysis checks whether the variable is protected by the same lock or not. The analysis of the two respectively reduces the false negatives and false positives. In addition, slicing principle combined with timing constraint graph is used in happens-before analysis. The timing constraint graph shows not only the occurring sequence of access events in each thread but also the interaction
between threads through directed edges. Slices for each access event can be obtained from the timing constraint graph. A certain condition of two slices is used to determine whether there is a happens-before relationship. HB Filter is used to avoid some false positives caused by the interaction of threads. The detection process is presented in Figure 2.

![Data race detection diagram](image)

Figure 2. Data race detection

Taken a concurrent program [19] presented in Figure 3 as an example, we show how our approach detects data race in the following section. In this example, threads $T_1$ and $T_2$ are created by thread `main`, and they call function `m1()` by overwriting function `run()`, in which a value is written to field $f$ of the static variable $q$ by function `m1()`.

```java
public class MainThread{
    static Obj p, q, x; //suppose q, x not alias
    public static void main(String args[]){
        ChildThread T1 = new ChildThread(q);
        ChildThread T2 = new ChildThread(x);
        T1.start(); T2.start();
        synchronized(p){
            m1(q);
            q.f=0;
        }
        T2.join( );
        x.f=200 ;
    }
    public static void m1(Obj y) { y.f=100 ; }
}
class ChildThread extends Thread{
    Obj a, b ;
    ChildThread(Obj o){ b =o;}
    public void run(){
        synchronized(o){
            MainThread.m1(b);
            m2( );
        }
    }
    public void m2(){
        obj z =new obj(...);
        z.g =... ;
    }
}
```

Figure 3. Sample example
3.1. Call Graph Analysis

Threads in program can be classified into two categories, main thread and child thread. For multithreaded programs, main thread executes from the function main(), while child thread uses function run() as entrance. We classify each function (intra-thread or inter-thread) by its execution entrance. We use Soot to construct the function call graphs. Figure 4 illustrates the process of call graphs construction. Methods from class ReachableMethods and TransitiveTargets of Soot are used to get all the called functions (Step 3). The purpose of Step 5 is used to distinguish whether the function is called directly or indirectly.

![Figure 4. The process of call graph analysis](image)

In Figure 3, function main( ) calls function start( ) of the child thread T1 (line 3), denoted by main( )→T1.start( ). Function run( ) in the child thread T1 calls function m1( ) (line 10), denoted by T1.run( )→m1( ). All call graphs of each thread can be found, and those call graphs without variable accesses will be filtered. For example, the call graph main( )→T2.join( ) (line 6) is filtered because there is no variable access operation in the called function T2.join( ). Table 1 describes function call graphs after filtering.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Function call graphs that exist variable access operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>main( ) // The entrance function of the main thread</td>
</tr>
<tr>
<td></td>
<td>main( )→m1( )</td>
</tr>
<tr>
<td>T1</td>
<td>T1.run( )→m1( )</td>
</tr>
<tr>
<td></td>
<td>T1.run( )→m2( )</td>
</tr>
<tr>
<td>T2</td>
<td>T2.run( )→m1( )</td>
</tr>
<tr>
<td></td>
<td>T2.run( )→m2( )</td>
</tr>
</tbody>
</table>

Note that there are only direct calls in Figure 3, while indirect calls exist in some programs. In other words, the entrance function calls a function after several function invocations, which is called the transitive target in Figure 4. For example, if the entrance function main( ) reaches the function m( ) after several calls, we denote it with main( )→m( ).

3.2. Intra-Thread Analysis

Variable access events in each thread can be obtained based on Table 1. The threadID can be obtained according to the call graph where the variable is located. We can judge whether the access operation is write according to the variable access operation, which is indicated by the keyword iswrite. We can determine that the variable access holds one or more locks based on the synchronization primitive. For a field, an analysis result of the access operation is recorded as VAi: threadID-field-iswrite-{lock1(*), lock2(*), ...}. For example, this is shown by the variable access events of the entrance function main( ) (lines 5 and 7). The code in Line 5 shows that the thread main writes the filed f of the static variable q and there is a lock synchronized(p), denoted by main-q.f-write-{lock(p)}. Similarly, if there is no lock, the variable access event (line 7) is denoted by main-x.f-write-null. All variable accesses in each thread are listed in Table 2. We record the ith variable access event as VAi.
Table 2. All variable accesses in each thread

<table>
<thead>
<tr>
<th>Thread</th>
<th>Function call graphs that exist variable access operations</th>
<th>Variable access</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>main()</td>
<td>VA1: main-q.f-write-{lock(p)}</td>
</tr>
<tr>
<td></td>
<td>main() \rightarrow m1()</td>
<td>VA2: main-x.f-write-{null}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VA3: main-q.f-write-{lock(p)}</td>
</tr>
<tr>
<td>T1</td>
<td>T1.run() \rightarrow m1()</td>
<td>VA4: T1-q.f-write-{lock(a)}</td>
</tr>
<tr>
<td>T2</td>
<td>T2.run() \rightarrow m1()</td>
<td>VA5: T1-z.g-write-{lock(a)}</td>
</tr>
<tr>
<td></td>
<td>T2.run() \rightarrow m2()</td>
<td>VA6: T2-x.f-write-{lock(a)}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VA7: T2-z.g-write-{lock(a)}</td>
</tr>
</tbody>
</table>

3.3. Inter-Thread Analysis

Race detection can be performed for each pair of access events between threads. For the detected races, alias analysis, happens-before analysis, timing constraint graph, and slicing principle are used to identify whether there exist false negatives or false positives.

3.3.1. Pre-Race Detection

Different threads accessing the same variable may lead to data race, which is called pre-race. Accessing events for the same variable between different threads are compared to detect pre-race. Figure 5 illustrates the process of pre-race detection. The code in Line 2 judges if access to the variable comes from different threads, and only different threads can produce pre-races. The code in Line 5 will filter these access event pairs that have unnecessary conditions for race.

```
Algorithm preRaceDetect(long threadID, SootField accessedField, boolean isWrite)
Input
  threadID // threadIDi and threadIDj of VAi-VAj (i ≠ j)
  accessedField // access objects of VAi-VAj (i ≠ j)
  isWrite // access operations of VAi-VAj (i ≠ j)
Output
  pre-races
Step
1  FOR each pair of variable accesses VAi-VAj (i ≠ j)
2  IF threadIDi == threadIDj
3  Filter this VAi-VAj // Two access events from the same thread
4  FOR the remaining pairs of variable accesses
5  IF (accessedField == accessedFieldj) \&\& (isWritei == isWritej)
6  Report VAi-VAj is a pre-race
```

Figure 5. The algorithm of pre-race detection

For example, pre-race is detected for VA1: main-q.f-write and VA4: T1-q.f-write (lock restriction is removed) in Table 2. It indicates that two threads (main and T1) write the same field f without protection and then a pre-race will be detected, denoted by VA1-VA4. All pairs of variable accesses that possibly produce pre-race are listed in Table 3. To facilitate subsequent analysis, we represent ith pre-race as Racei.

Table 3. The description of all pairs of variable accesses that possibly produce pre-race

<table>
<thead>
<tr>
<th>Pre-Race</th>
<th>main - T1</th>
<th>main - T2</th>
<th>T1 - T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Race1</td>
<td>VA1-VA4</td>
<td>VA2-VA6</td>
<td>VA5-V7</td>
</tr>
<tr>
<td>Race2</td>
<td>VA2-VA4</td>
<td>VA3-VA6</td>
<td>VA7-V8</td>
</tr>
<tr>
<td>Race3</td>
<td>VA3-VA4</td>
<td></td>
<td>Jt2-V2</td>
</tr>
</tbody>
</table>

3.3.2. Alias Analysis

For each pre-race, we leverage an alias analysis that contains alias variable analysis and alias lock analysis. The aim of alias variable analysis is to detect if the same access variable of different threads is in the same memory location or not. If two access variables use the different name and the memory location is the same, it may be a true data race. The aim of alias lock analysis...
analysis is to detect if the pair of access events are protected by the same lock or not; if not, the pre-race may be true.

We conduct alias variable analysis for pre-race: \(VA_i-VA_j\), which checks if the two access objects of \(VA_i-VA_j\) are aliases or not. If they are, the same memory in which the variables are located are accessed by two threads. For example, in Race2: \(VA_2-VA_4\), the access objects are \(x.f\) and \(q.f\), respectively. Since \(x\) and \(q\) are not aliases in Figure 3, \(x.f\) and \(q.f\) point to different memory locations, so Race2 is a fake race. However, suppose that \(x\) and \(q\) are a pair of aliases, the access objects \(x.f\) and \(q.f\) of thread main and \(T1\) will be located in the same memory, and Race2 may be a real data race. In this case, if there is no alias variable analysis for \(x\) and \(q\), then \(x.f\) and \(q.f\) will be treated as two different access objects in the detection process, which may lead to a false negative about Race2. It can be seen that the alias variable analysis can reduce the false negatives caused by ignoring the alias phenomenon of variables.

There also exists an alias phenomenon in the lock set analysis, and the alias lock analysis can reduce the false positives. For example, alias lock analysis collects the lock information of Race3(\(VA_3-VA_4\)) to get \(<\text{lock}(p), \text{lock}(a)>\). Suppose \(p\) and \(a\) are a pair of aliases, and \(<\text{lock}(p), \text{lock}(a)>\) is a pair of alias lock, which is equivalent to two access events being protected by the same monitor. As a result, the pre-race is fake to be eliminated. If the alias lock is not considered, then the intersection of the lock sets of the two access events is null, and the pre-race is likely to be reported as a real race, which will be a false positive. Therefore, the alias lock analysis can avoid some false positives and improve the accuracy of data race detection. In addition, Race8 is a fake race to be eliminated because the access events of field \(g\) are protected by the same monitor \(\text{lock}(a)\) in \(VA_5-VA_7\). We recognize three data races after alias analysis, including Race1, Race3, and Race5.

3.3.3. Timing Constraint

Lock is essentially a kind of timing constraint when considering access operations. In addition, the interaction between threads can also lead to happens-before relationship. For example, \texttt{start} / \texttt{join} primitives can generate a sequence for variable accesses between threads. We analyze happens-before relationships by constructing timing constraint graphs and using program slicing.

Timing constraint graphs represent the sequence of access events. Figure 6 shows the constraint graph of the sample program in Figure 3. Each node represents one execution of the statement, and the directed edge between the nodes represents the execution sequence. The statements of each thread are connected. There is a sequential relationship between threads due to \texttt{start} / \texttt{join} primitives. We denote access objects as underlined in Figure 6.

![Timing constraint graph](image)

Starting from a subset of a program’s behavior, slicing reduces that program to a minimal form that still produces that behavior [20]. A slicing criterion of a program \(P\) is a tuple \(<s, v>\), where \(s\) is a statement in \(P\) and \(v\) is the variable in statement \(s\). The slice \(S\) of the program \(P\) is an executable program, and \(S\) is composed of all statements in program \(P\) that may affect the value of the variable \(v\) at the statement \(s\).

Supposing that two slices for the same variable \(v\) at the statement \(si\) and \(sj\) are \(Si\) and \(Sj\) respectively, there exists a happens-before relation if and only if \((si \in Sj) \land (sj \notin Si) (i \neq j)\). A slice of an access event can be obtained from the relation of directed edges in Figure 6. For example, we do slicing analysis for Race5: \(VA_2-VA_6\). The slice of \(VA_2\) (Statement \(s_2\) for \(VA_2\) in Line 7 in Figure 3) is statically sliced backwards based on the slicing criterion \(<7, f>\) to get the slice \(S_2: <7>\), while the slice of \(VA_6\) (Statement \(s_6\) for \(VA_6\) in Line 8 of thread \(T2\) in Figure 3) is statically sliced backwards based on the slicing criterion \(<8, f>\) to get the slice \(S_6: <8, 7>-\). Due to \((s_2 \notin S_6)\land (s_6 \notin S_2)\), the Race5 with the happens-before is eliminated as a fake race. Similarly, we conduct the analysis for the remaining Race1 and Race3, which can be identified as two data races because there is no happens-before relation.
The timing constraint analysis can determine the happens-before relationship between the access events, reduce the false positives, and improve the accuracy of detection.

4. Evaluation

4.1. Benchmarks

We evaluate our approach on six benchmarks, which cover different sizes of concurrent programs. These benchmarks are selected from two benchmark suites, JGF [21] and IBM Contest benchmark suite [22]. We select three benchmarks, raytracer, moldyn, and sor, from the JGF benchmark suite, and two benchmarks, critical and mergesort, from the IBM Contest Benchmark suite. These benchmarks have been configured with four working threads and the maximum data size. The configuration of these benchmarks is shown in Table 4.

Table 4. The configuration of benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Thread</th>
<th>#Class</th>
<th>Byte code(B)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytracer</td>
<td>4</td>
<td>14</td>
<td>46890</td>
<td>3D ray tracer</td>
</tr>
<tr>
<td>moldyn</td>
<td>4</td>
<td>4</td>
<td>23352</td>
<td>molecular dynamics simulation</td>
</tr>
<tr>
<td>sor</td>
<td>4</td>
<td>2</td>
<td>8501</td>
<td>successive over-relaxation algorithm</td>
</tr>
<tr>
<td>account</td>
<td>3</td>
<td>2</td>
<td>4096</td>
<td>A bank account service algorithm</td>
</tr>
<tr>
<td>mergesort</td>
<td>5</td>
<td>2</td>
<td>13722</td>
<td>algorithm for merging sort</td>
</tr>
<tr>
<td>MainThread</td>
<td>3</td>
<td>3</td>
<td>4154</td>
<td>sample example of Figure 3</td>
</tr>
</tbody>
</table>

All experiments were conducted on a 16-core 2.60GHz Intel Xeon E5-2650 workstation with 128GB RAM. The workstation runs Windows 7 operating system, with Eclipse4.5.1 and JDK 1.8.0_31 installed. We run every benchmark five times, remove the maximum and minimum, and obtain the average value.

4.2. Experimental Results

We compare our results with RVPredict [23], a race detection tool based on causality-preserving predictive analysis of which the tool is available. A key insight of RVPredict is the inclusion of abstracted control flow information into the execution model, which increases the space of the causal model permitted by classical happens-before.

In the timing constraint analysis of our method, control flow information is used and abstracted into the timing constraint graph to determine the happens-before relationship between access events. Table 5 presents the results between SRD and RVPredict.

Table 5. The experiment result

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Real Races</th>
<th>Detected Races</th>
<th>SRD Detection time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytracer</td>
<td>1</td>
<td>4</td>
<td>1.93</td>
</tr>
<tr>
<td>moldyn</td>
<td>0</td>
<td>0</td>
<td>1.72</td>
</tr>
<tr>
<td>sor</td>
<td>0</td>
<td>0</td>
<td>1.08</td>
</tr>
<tr>
<td>account</td>
<td>Not reported</td>
<td>5</td>
<td>1.02</td>
</tr>
<tr>
<td>mergesort</td>
<td>Not reported</td>
<td>9</td>
<td>1.57</td>
</tr>
<tr>
<td>MainThread</td>
<td>2</td>
<td>2</td>
<td>0.93</td>
</tr>
</tbody>
</table>

For the raytracer benchmark, the number of data races detected by RVPredict is four, while SRD finds only one. In fact, this benchmark only has one harmful data race. SRD is able to find these potential races and reserve those harmful races, which shows that the detection of SRD is accurate.

For the mergesort benchmark, SRD detects 15 data races while RVPredict only reports nine data races. We manually check data races reported by SRD and have confirmed that all the data races reported by SRD are real and valid. SRD is able to find potential races and has a high detection coverage, because the execution trace of access events is uncertain when two access events is not connected by directed edges in the timing constraint graph of SRD. An execution trace is abstracted as a sequence of events in RVPredict, which is likely to miss some traces and lead to false positives.

For the moldyn, sor, account, and MainThread benchmarks, both SRD and RVPredict have the same detection rate and both can report positive races.

In general, RVPredict finds 20 races in total, whereas SRD reports 23 races. The races detected by RVPredict are a subset of SRD. In addition, it can be seen that the detection time of the SRD is within the acceptable range.
5. Related Work

There are many analysis works in data race detection. Compared with dynamic analysis, static analysis has the advantages of detection coverage and low overhead. Initial attempts of race detection can be classified into two categories: the lockset and happens-before approaches. The lockset-based approach considers only lock mutual exclusion, so it is unsound. The happens-before approach depends on the specific execution trace, which may lead to false negatives. Therefore, hybrid approaches have been developed.

Ali et al. [24] proposed a dynamic race detection tool named Helgrind+ that incorporates correct handling of condition variables and a combination of the lockset algorithm and happens-before relation. The happens-before relationship is obtained through thread segment, which splits the execution sequence of a thread. Each shared memory unit can only retain the latest thread segment, and the lockset algorithm determines whether the race is formed. Helgrind+ reduces the number of both false negatives (missed races) and false positives.

Serebryany et al. [16] described the hybrid algorithm ThreadSanitizer (based on happens-before and locksets) used in the detector. Unlike Hengrind+, ThreadSanitizer saves all potential concurrent writing segment sets and all potential concurrent reading segment sets that happen after the writing segment. ThreadSanitizer introduces dynamic annotations, a type of race detection API that allows a user to inform the detector about any tricky synchronization in the user program.

Acculock [17] is the first hybrid race detector to balance precision and coverage by leveraging the light-weight epoch clocks. Acculock analyzes programs by reasoning about the subset of the happens-before relation observed with lock acquires and releases excluded, thereby reducing its sensitivity to thread interleaving. When such a weaker happens-before relation is violated, Acculock applies a new efficient lockset algorithm to enforce a lock-based synchronization discipline by distinguishing the locks protecting reads and writes. Since this approach uses the epoch and lockset related to the last read and write of the shared memory, there are false positives and false negatives.

Choi et al. [25] is the first static tool to conduct automatic analysis of object-oriented concurrent programs without using type analysis. It takes the access event as the center and makes the alias analysis of the specific traces and all traces to obtain the real race and the possible race. However, it does not abstractly analyze the start/join primitives that generate the happens-before relationship.

Our approach is similar to Wu's work [19], which resolves the data race problem by cross-threaded control flow analysis, applies context-sensitive and flow-sensitive alias analysis, and analyzes the happens-before relationship through the timing constraint graph. However, the analysis of [19] is at the object level, and our approach focuses on the access events. In addition, we analyze the intra-/inter-thread control flow, and our alias analysis not only includes alias variable analysis but also alias lock analysis.

6. Conclusion

This paper presents a static data race detection methodology and tool that use Soot to conduct intra- and inter-thread analysis to construct function call graphs. SRD focuses on variable access events, and these access events are collected according to call graphs. Inter-thread analysis uses slicing principle combined with timing constraint graph to get static slices of access events, and it determines happens-before relationships. Furthermore, the alias variable and alias lock on data race are considered in order to reduce false negatives and false positives in the detection process. Experimental results have shown that SRD can obtain good detection results when it is applied to several benchmarks from Java Grande and the IBM Contest benchmark suite. Future works will focus on conducting tests and analyses on large-scale practical applications.

Acknowledgement

This work was partly financially supported by the National Natural Science Foundation of China (No. 61440012), the Natural Science Foundation of Hebei Province (No. F2016208007), and the Fundamental Research Foundation of Hebei Province (No. 18960106D). The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers.

References


Yang Zhang received his Ph.D. from the Beijing Institute of Technology. Currently, he is an associate professor at Hebei University of Science and Technology. His research interests include parallel programming, software refactoring, and program analysis.

Yanan Liang is currently a candidate for a Master’s degree at Hebei University of Science and Technology. Her research interests focus on parallel programming and data race detection.

Dongwen Zhang received her Ph.D. from the Beijing Institute of Technology. She is currently a professor in the School of Information Science and Engineering at Hebei University of Science and Technology. Her research interests focus on parallel programming models and software refactoring for parallelism.